IMPLEMENTASI *LOAD BALANCING WEB* SERVER DENGAN ALGORITME *WEIGHTED LEAST CONNECTION* PADA *SOFTWARE DEFINED NETWORK*

SKRIPSI

Untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun oleh:
Hafizhul Karim
NIM: 135150207111076

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA
MALANG
2018

PENGESAHAN

IMPLEMENTASI *LOAD BALANCING WEB* SERVER DENGAN ALGORITME *WEIGHTED LEAST CONNECTION* PADA *SOFTWARE DEFINED NETWORK*

SKRIPSI

Diajukan untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun Oleh :
Hafizhul Karim
NIM: 135150207111076

Skripsi ini telah diuji dan dinyatakan lulus pada
30 Juli 2018
Telah diperiksa dan disetujui oleh:

Dosen Pembimbing I	Dosen Pembimbing II
Rakhmadhany Primananda, S.T, M.Kom	Widhi Yahya, S.Kom., M.Sc.
NIK: 2016098604061001	NIK: 2016078911211001

Mengetahui
Ketua Jurusan Teknik Informatika

Tri Astoto Kurniawan, S.T, M.T, Ph.D
NIP: 19710518 200312 1 001

PERNYATAAN ORISINALITAS

Saya menyatakan dengan sebenar-benarnya bahwa sepanjang pengetahuan saya, di dalam naskah skripsi ini tidak terdapat karya ilmiah yang pernah diajukan oleh orang lain untuk memperoleh gelar akademik di suatu perguruan tinggi, dan tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang lain, kecuali yang secara tertulis disitasi dalam naskah ini dan disebutkan dalam daftar pustaka.

Apabila ternyata didalam naskah skripsi ini dapat dibuktikan terdapat unsur-unsur plagiasi, saya bersedia skripsi ini digugurkan dan gelar akademik yang telah saya peroleh (sarjana) dibatalkan, serta diproses sesuai dengan peraturan perundang-undangan yang berlaku (UU No. 20 Tahun 2003, Pasal 25 ayat 2 dan Pasal 70).

Malang, 30 Juli 2018

Hafizhul Karim

NIM: 135150207111076

KATA PENGANTAR

Assalamualaikum Warrahmatullahi Wabarakatuh. Dengan menyebut nama Allah Yang Maha Pengasih dan Maha Penyayang. Segala puji bagi Allah karena atas berkah, rahmat dan hidayah-Nya penulis dapat menyelesaikan skripsi yang berjudul "Implementasi *Load Balancing Web* Server dengan Algoritme *Weighted Least Connection* pada *Software Defined Network*" dengan baik.

Penelitian skripsi ini mendapat banyak bantuan dari berbagai pihak yang terus memberikan bibingan, saran, dukungan, motivasi maupun doa. Oleh karena itu, penulis mengucapkan terima kasih kepada :

1. Kepada orang tua yang sangat saya sayangi, bapak Rapitos Putranto Baharudin dan ibu Haryanti Azhar dan serta seluruh keluarga atas dukungan dan kasih sayang yang selalu diberikan.
2. Rakhmadhany Primananda, S.T, M.Kom, sebagai Dosen Pembimbing pertama, atas segala bimbingan, waktu, kritik dan saran yang diberikan kepada penulis selama penelitian skripsi.
3. Widhi Yahya, S.Kom., M.Sc., sebagai Dosen Pembimbing kedua, atas segala bimbingan, waktu, kritik dan saran yang diberikan kepada penulis selama penelitian skripsi.
4. Seluruh Bapak dan Ibu dosen Fakultas Ilmu Komputer Universitas Brawijaya atas segala bimbingan dan ilmu yang diberikan kepada penulis.
5. Seluruh sahabat-sahabat saya Ashraff D'Bryff, Mukhammad Sharif Hidayatulloh, Frondy Fernanda, Roliand Prasetya, Imron Hari, Ardiansyah Setiajati, Auliaur Rasyid, Ranu Goldan, Faizal Putra, Nurwida Mariatul, Annisya Aprilia, Feri Angga.
6. Semua pihak yang telah membantu kelancaran penelitian skripsi dan memberikan doa yang tidak dapat disebutkan penulis satu persatu.

Semoga semua jasa dan amal baik yang diberikan kepada penulis mendapatkan balasan dari Allah Subhanahu wa ta'ala. Penulis menyadari bahwa penelitian ini masih terdapat kekurangan dan kesalahan yang disebabkan oleh keterbatasan ilmu. Oleh karena itu, penulis mengharapkan kritik dan saran kepada pembaca sebagai perbaikan pada penelitian selanjutnya. Semoga laporna skripsi ini bermanfaat bagi semua pihak, Amin.

Malang, 30 Juli 2018

Penulis

hafis.karim5@gmail.com

ABSTRAK

Permasalahan pada *web* server ketika *request* yang dilakukan oleh *user* sangat banyak sehingga beban kerja server semakin meningkat, dapat memungkinkan server mengalami *down*. Oleh karena itu dibutuhkan mekanisme pembagian beban pada *web* server menggunakan teknik *load balancing*. *Load balancing* merupakan sebuah teknik untuk membagi beban kerja pada dua atau lebih server ketika ada *request* dari *client*, hal ini bertujuan agar trafik berjalan secara optimal. Pada *load balancing* terdapat beberapa algoritme, antara lain *weighted least connection*, *least connection, round robin,* dan lain-lain. Pada penelitian ini menggunakan algoritme *weighted least connection*. Algoritme *weighted least connection* membagi beban server berdasarkan nilai *weight* (bobot). Arsitektur *Software Defined Network* (SDN) bersifat *programmable* dan memberikan fasilitas kepada *Network Administrator* untuk merancang dan mengimplementasikan strategi *load balancing* tanpa keterbatasan vendor. Selain itu penerapan SDN merupakan sebuah solusi dalam perkembangan dari kebutuhan jaringan yang semakin kompleks yaitu dalam segi penekanan biaya untuk investasi perangkat keras jaringan. Oleh karena itu arsitektur dengan pendekatan SDN sangat diperlukan. Untuk implementasi menggunakan mininet sebagai simulator dan *controller* yang dipakai menggunakan POX. Parameter yang digunakan untuk pengujian yaitu *connection rate, throughput,* dan CPU *usage*. *Tool* yang digunakan untuk pengujian yaitu *httperf*. Hasil dari pengujian *connection rate, response time,* dan *throughput* dengan memberikan *request* dari *client* sebanyak 3000 dengan *rate* 75, 15, dan 300 yaitu nilai *connection rate* tertinggi sebesar 296,28 Conn/s, nilai *response time* terkecil yaitu 2,34 ms dan nilai *throughput* tertinggi sebesar 805,66 KB/s. Dan yang terakhir hasil dari pengujian CPU *usage*, server 1 memiliki nilai 21,6%, server 2 memiliki nilai 24,6%, dan server 3 memiliki nilai 26,3%.

Kata kunci: *Software Defined Network* (SDN), *Load Balancing, Weighted Least Connection,* Bobot.

ABSTRACT

The Problem of the web server when the number of requests used by the user so that the workload on the server increases quickly resulting in the server down. Therefore we need load sharing mechanism on web server using load balancing technique. Load balancing is a technique to distribute incoming traffic between available servers so that requests can be handled and respond faster. In load balancing there are several algorithms, including weighted least connection, least connection, round robin, and others. This research uses weighted least connection algorithm. The weighted least connection algorithm divides the server load based on the weight value. Software Defined Network Architecture (SDN) is programmable and provides facilities to the Network Administrator to design and implement load balancing strategies without vendor limitations. In addition to that the application of SDN is a solution in the development of the increasingly complex network needs that is in terms of cost suppression for network hardware investment. Therefore, the architecture with the SDN approach is needed. For implementation use mininet as simulator and controller used with POX. Parameters used for testing are connection rate, throughput, and CPU usage. The tool used for testing is httperf. The result of connection rate and throughput testing by giving client request 3000 with the rate of 75, 150, and 300 with the highest connection rate is 296,28 Conn/s, the lowest response time is 2,34 ms and the highest throughput value is 805,66 KB/s. And the last result of testing CPU usage, server 1 has a value of 21.6%, server 2 has a value of 24.6%, and server 3 has a value of 26.3%.

Keywords: *Software Defined Network* (SDN), *Load Balancing, Weighted Least Connection, Weight.*

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

BAB 1 PENDAHULUAN

1.1 Latar belakang

Permasalahan pada *web* server ketika *request* yang dilakukan oleh *user* sangat banyak sehingga beban kerja server semakin meningkat, dapat memungkinkan server mengalami *down*. Hal ini terjadi karena *request* yang ditangani oleh satu *web* server. Oleh karena itu dibutuhkan *web* server dengan pendekatan *multiple* server. Fungsi dari pendekatan *multiple* server yaitu sistem *web* server direplika untuk meningkatkan waktu respon (Singh & Kumar, 2011). Pada *multiple* server dibutuhkan mekanisme pembagian beban supaya kinerja *web* server tetap stabil. Mekanisme pembagian beban menggunakan teknik *load balancing*.

Load balancing merupakan sebuah teknik untuk membagi beban kerja pada dua atau lebih server ketika ada *request* dari *client*, hal ini bertujuan agar trafik berjalan secara optimal (Rahman, et al., 2014). Dalam teknik *load balancing* terdapat beberapa algoritme antara lain, *round robin, weighted round robin, least connection, weighted least connection,* dan lainnya. Algoritme *weighted least connection* hampir sama dengan *least connection,* hanya saja algoritme ini menentukan bobot kinerja dari setiap server (Red Hat, 2014). Setiap server akan diberikan nilai bobot atau *weight*, semakin tinggi nilai bobot yang diberikan pada server maka server akan menerima request lebih banyak dari server yang memiliki nilai bobot rendah. Pada algoritme ini diterapkan pada server yang heterogen. Server heterogen memiliki spesifikasi yang berbeda antara server satu dengan server lainnya, contoh server 1 mempunyai jumlah *core* sebanyak 2 dan server 2 mempunyai jumlah *core* sebanyak 3. Algoritme *least connection* bekerja melakukan pembagian *request* berdasarkan banyaknya koneksi yang sedang dilayani oleh server, lalu server yang melayani koneksi dengan jumlah paling sedikit akan diberikan beban untuk *request* berikutnya (Supramana & Prismana, 2016). Pada pengujian beban statis, algoritme *weighted least connection* lebih baik dalam mendistribusikan permintaan HTTP, balasan HTTP, dan *throughput*, namun waktu tanggap atau *response time* lebih lambat jika dibandingkan dengan algoritme *least connection* (Angsar, 2014). Hal ini terjadi karena algoritme *weighted least connection* dapat menampung jumlah koneksi yang berbeda, sedangkan algoritme *least connection* dapat menampung jumlah koneksi yang sama (Rahmana, 2017).

Load balancing dengan arsitektur *software defined network* (SDN) bersifat *programmable* dan memberikan fasilitas kepada *Network Administrator* untuk merancang dan mengimplementasikan strategi algoritme *load balancing* tanpa keterbatasan vendor (Sabiya & Singh, 2016). *Software defined network* (SDN) merupakan sebuah arsitektur yang memisahkan *control plane* dan *data plane* pada perangkat jaringan seperti *switch* dan *router*, sedangkan pada jaringan tradisional *control plane* dan *data plane* ini digabung menjadi satu pada perangkat jaringan. *Control plane* berfungsi untuk mengatur logika jaringan dan *data plane* berfungsi untuk mengatur bagaimana paket akan diteruskan pada *hop* berikutnya

(Kreutz, et al., 2014). Selain itu penerapan SDN merupakan sebuah solusi dalam perkembangan dari kebutuhan jaringan yang semakin kompleks yaitu dalam segi penekanan biaya untuk investasi perangkat keras jaringan. (Paul Göransson, 2014). Oleh karena itu arsitektur dengan pendekatan SDN sangat diperlukan.

Pada penelitian sebelumnya terkait tentang *load balancing* dengan pendekatan *software defined network* (SDN) sudah pernah diterapkan salah satunya penelitian yang dilakukan oleh Sabiya dan Japinder Singh yang berjudul *"Weighted Round-Robin Load Balancing Using Software Defined Network"*. Penelitian tersebut menggunakan server heterogen. Pada masing-masing server diberikan nilai weight. Server 1 diberikan nilai *weight* 1, server 2 diberikan nilai *weight* 3, dan server 3 diberikan nilai *weight* 6. Hasil yang didapatkan yaitu algoritme *weighted round robin* bekerja dengan cukup baik dalam pengalokasian beban atau *request client* yang hampir merata pada setiap server (Sabiya & Singh, 2016). Berdasarkan dari penelitian sebelumnya yang menjadi pembeda dalam penelitian ini yaitu algoritme *load balancing* yang digunakan *weighted least connection*.

Oleh karena itu penelitian ini menerapkan algoritme *weighted least connection* karena pada setiap server memiliki nilai bobot yang berbeda dan menggunakan server heterogen. Penentuan nilai bobot (*weight*) ditentukan oleh *administrator*. *Administrator* dapat mengkonfigurasi server mana yang dapat menampung koneksi yang banyak dan sedikit. Nilai bobot secara *default* bernilai 1, tidak boleh bernilai 0. Pada algoritme *weighted least connection* nilai bobot disesuaikan dengan spesifikasi server, misalkan terdapat 2 server yang memiliki spesifikasi berbeda. Server 1 memiliki spesifikasi lebih unggul dibanding server 2, maka nilai bobot pada server 1 lebih besar dari server 2. Arsitektur jaringan yang digunakan yaitu *software defined network* karena memberikan manajemen jaringan yang dinamis dan kemudahan dalam melakukan konfigurasi jaringan yang dapat dilakukan terpusat pada *controller*. Dengan melakukan penelitian ini, diharapkan penggunaan algoritme *weighted least connection* dapat menjadikan kinerja *web* server yang lebih baik.

1.2 Rumusan Masalah

Pada rumusan masalah ini dilakukan berdasarkan latar belakang diatas yang dapat dirumuskan dalam penelitian ini, antara lain:

1. Bagaimana cara implementasi *load balancing web* server dengan algoritme *weighted least connection* pada *software defined network*?
2. Bagaimana kinerja *web* server saat diterapkan *load balancing* menggunakan algoritme *weighted least connection* pada *software defined network* dengan parameter *connection rate, response time, throughput,* dan CPU *usage*?

1.3 Tujuan

Tujuan dari penelitian ini berdasarkan latar belakang diatas, antara lain:

1. Untuk melakukan implementasi *load balancing web* server dengan algoritme *weighted least connection* pada *software defined network.*
2. Untuk mengetahui performa *web* server dengan algoritme *weighted least connection* pada *software defined network* dengan parameter *connection rate*, *response time, throughput,* dan CPU *usage*.

1.4 Manfaat

Manfaat dari penelitian ini diharapkan mempunyai banyak manfaat terutama penerapan teknik *load balancing* menggunakan algoritme *weighted least connection* dan diharapkan menjadikan performa *web* server yang lebih baik.

1.5 Batasan Masalah

Agar penelitian ini tidak mengambang jauh dari tujuan yang sudah ditentukan maka berdasarkan permasalahan yang sudah dirumuskan sebelumnya, ada beberapa batasan masalah yaitu:

1. Penelitian ini difokuskan pada implementasi *load balancing web* server dengan menggunakan algoritme *weighted least connection* pada *software defined network.*
2. Menggunakan bahasa pemrograman python
3. Menggunakan *controller* pox.
4. Menggunakan mininet server sebagai *web* server.
5. Melakukan pengujian performa server dengan melakukan *request client* ke *web* server dan menggunakan jumlah *rate.*
6. Pengujian pada penelitian ini berlangsung sebanyak 5 kali yang nantinya akan diambil nilai rata-rata dari setiap parameter.

1.6 Sistematika Pembahasan

Sistematika pembahasan dilakukan untuk memberikan gambaran langkah-langkah dari penelitian ini yang meliputi beberapa bab, yaitu:

BAB I PENDAHULUAN

Pada bab ini menjelaskan latar belakang, rumusan masalah, tujuan, manfaat, batasan masalah, dan sistematika pembahasan.

BAB II LANDASAN KEPUSTAKAAN

Pada bab ini menjelaskan tentang kajian pustaka dan dasar-dasar teori mengenai teknologi jaringan *software defined network* (SDN) dan *load balancing.*

BAB III METODOLOGI

Pada bab ini menjelaskan tentang metode yang digunakan dalam penelitian yang diantaranya studi literatur, analisis kebutuhan

sistem, perancangan sistem, implementasi sistem, pengujian sistem, dan kesimpulan.

BAB IV REKAYASA KEBUTUHAN

Pada bab ini menjelaskan tentang deskripsi sistem secara umum, analisis kebutuhan fungsional dan non-fungsional yang meliputi kebutuhan perangkat lunak dan perangkat keras.

BAB V PERANCANGAN DAN IMPLEMENTASI

Pada bab ini membahas tahap-tahap perancangan sistem yang dilakukan untuk penerapan *load balancing* di *web* server menggunakan algoritme *weighted least connection* pada *software defined network*, menggunakan *pox* sebagai *controller* dan *mininet* sebagai simulasi.

BAB VI PENGUJIAN

Pada bab ini membahas proses pengujian dan analisa pada sistem *load balancing software defined network.*

BAB VII PENUTUP

Pada bab ini membahas kesimpulan dan saran yang di dapat dari hasil pengujian sistem *load balancing software defined network.*

BAB 2 LANDASAN KEPUSTAKAAN

2.1 Tinjauan Pustaka

Pada penelitian ini tinjaun pustaka diambil dari penelitian sebelumnya yang berkaitan tentang load balancing pada *software defined network* (SDN) dan dijadikan sebagai pendukung untuk melaksanakan penelitian. Berikut ini tabel kajian pustaka:

Tabel 2.1 Kajian Pustaka

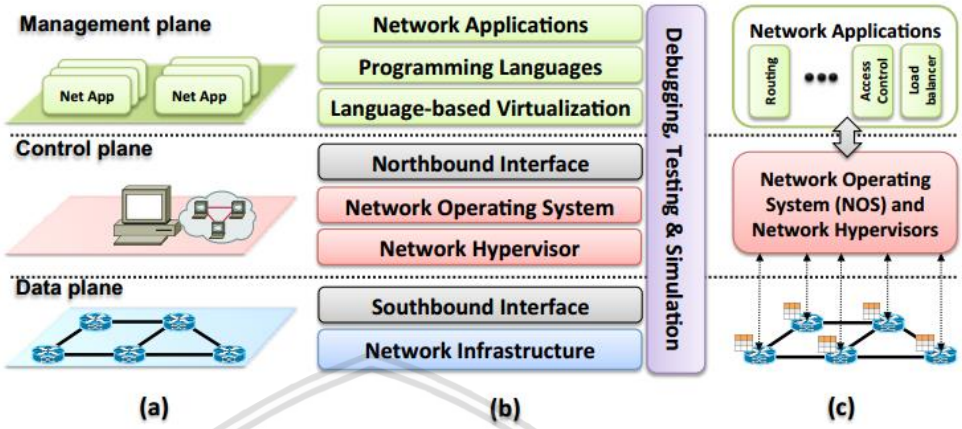
No	Nama penulis, Tahun dan Judul	Persamaan	Perbedaan	
			Penelitian Terdahulu	Rencana Penelitian
1	Sabiya, Japinder Singh (2016) *Weighted Round-Robin Load Balancing Using Software Defined Network*	Melakukan penerapan sistem *load balancing web* server pada *software defined network*	Menjelaskan implementasi sistem *load balancing* dengan algoritme *weighted round-robin* pada *software defined network*	Mengimplementasikan sistem *load balancing* dengan algoritme *weighted least connection* pada *software defined network*
2	Dany Rahmana (2017) Analisis *Load Balancing* Pada *Web* Server Menggunakan Algoritme *Weighted Least Connection*	Menggunakan algoritme *weighted least conection*	Menjelaskan implementasi sistem *load balancing* dengan algoritme *weighted least connection* pada jaringan tradisional	Mengimplementasikan sistem *load balancing* dengan algoritme *weighted least connection* pada *software defined network*

Berdasarkan pada Tabel 2.1 menjelaskan ada beberapa teori yang akan dijelaskan atau diulas lebih jauh untuk menganalisis, mengimplementasikan *load balancing* di *web* server dengan algoritme *weighted least connection* pada *software defined network*.

2.2 Software Defined Network

Software Defined Network (SDN) merupakan sebuah arsitektur yang memisahkan *network control/control plane* dan *forwarding* function*/data plane* yang memungkinkan *network control* menjadi program yang dapat diprogram secara langsung dan infrastruktur yang mendasarinya akan diabstraksikan untuk aplikasi dan layanan jaringan (Kreutz, et al., 2014). Manfaat dari SDN adalah

arsitektur baru yang dinamis, mudah dikelola, hemat biaya, dan mudah beradaptasi sehingga ideal untuk *bandwidth* yang besar dan dinamis dari aplikasi saat ini.



Gambar 2.1 Arsitektur *Software Defined Network*

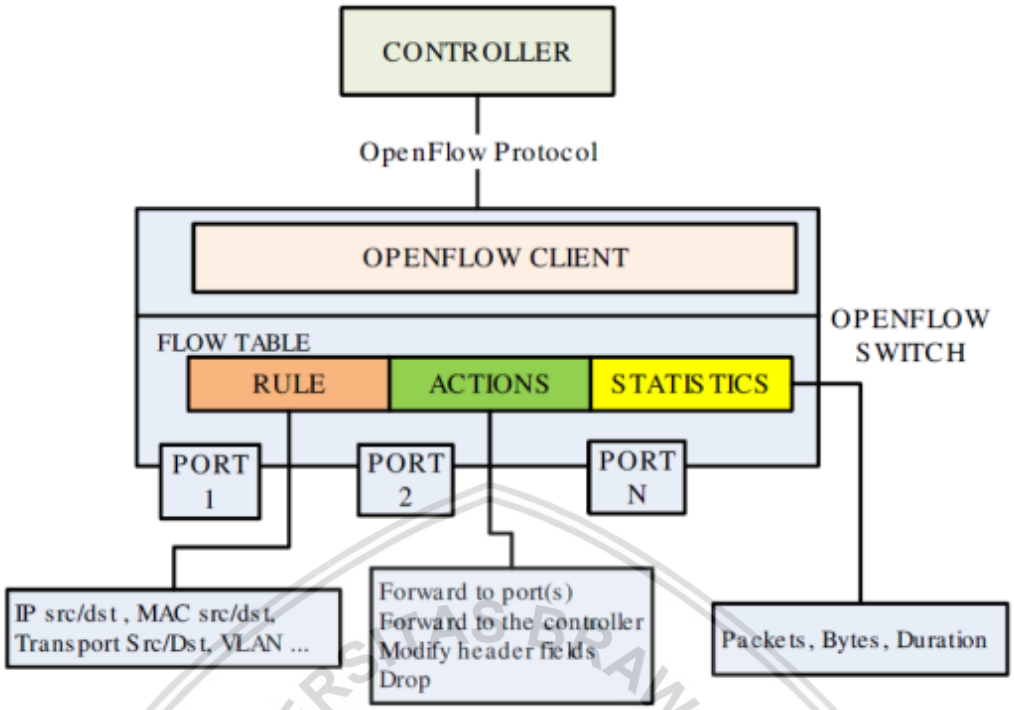
Sumber : (Kreutz, et al., 2014)

Pada Gambar 2.1 Arsitektur SDN terbagi menjadi 3 lapis/layer yaitu:

1. Layer Infrastruktur (*data plane*), terdiri dari elemen jaringan yang dapat mengatur atau mengontrol SDN *Datapath* sesuai dengan instruksi yang diberikan melalui *Control-Data-Plane Interface* (CDPI).
2. Layer Kontrol (*control plane*), terdiri dari entitas kontrol (*SDN Controller*) yang mentranlasikan kebutuhan aplikasi dengan infrastuktur dan memberikan instruksi yang sesuai untuk *SDN Datapath* serta memberikan informasi yang relevan dan dibutuhkan oleh *SDN Application.*
3. Layer Aplikasi (*application plane*), berada pada lapisan teratas yang berfungsi untuk komunikasi dengan sistem via *NorthBound Interface* (NBI).

2.2.2 Openflow

Openflow merupakan komunikasi antara *control plane* dan *data plane* pada arsitektur *software defined network* (Lee, 2014). *Openflow* adalah protokol standar *Open Network Foundation* (ONF) untuk mengontrol tabel *forwarding switch* atau *router* jarak jauh. Kontroler Openflow bagian dari kontroler SDN yang memberikan instruksi melalui channel ke switch Openflow. Sebuah switch Openflow mengelola beberapa tabel flow untuk menangani dan meneruskan paket ke tujuan.



Gambar 2.2 Arsitektur Protokol *Openflow*

Sumber : (Astuto, et al., 2014)

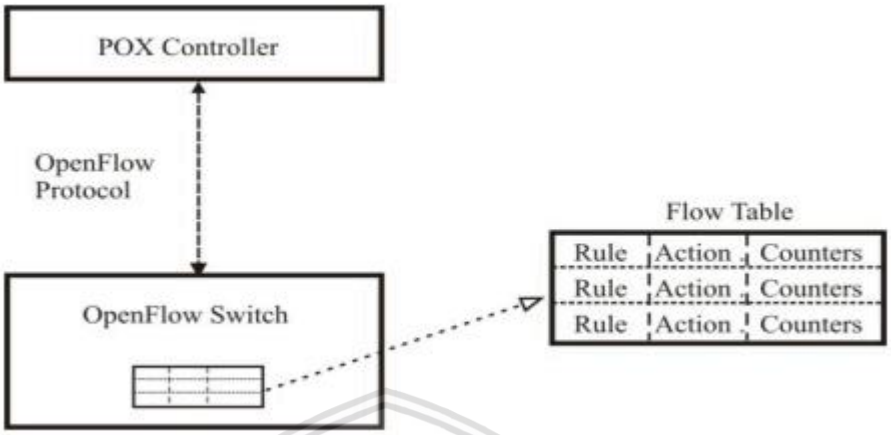
Pada Gambar 2.2 menjelaskan tentang Arsitektur protokol *openflow*. Entri dalam tabel *flow* terdiri dari sebuah aturan yang digunakan untuk mencocokan paket yang masuk, yaitu sebuah paket yang sesuai dengan peraturan. Pencocokan *file* mungkin berisi informasi yang ditemukan di paket *header*, *port* masuk, dan *metadata*. Tindakan harus diterapkan pada pencocokan dan mendikte bagaimana menangani paket yang sesuai. *Counter* digunakan untuk mengumpulkan statistik untuk *flow* tertentu seperti jumlah paket yang diterima. Sebuah *openflow switch* dapat membuat, memperbarui dan menghapus tabel entri pada tabel *flow* sesuai intruksi yang diberikan melalui protokol *openflow* dari kontroler *openflow*.

2.2.3 Controller

Controller merupakan salah satu bagian dari arsitektur jaringan *software defined network* yang mengelola *flow control* ke *switch*/*router* via *southbound* APIs dan aplikasi logika bisnis via *northbound* APIs untuk menyebarkan jaringan cerdas. Intinya controller pada SDN berfungsi untuk melakukan komunikasi dan konfigurasi antara *application layer* dan *infrastructure layer*. *Controller* dapat mengatur semua kebijakan yang ada pada jaringan seperti *routing*, *switching*, dan juga pengaturan jaringan lainnya. Oleh karena itu tugas *controller* pada *software defined network* sangatlah penting. Ada banyak sekali *controller* pada SDN diantaranya yaitu opendaylight, floodlight, POX, RYU, dan trema.

Berdasarkan pada Gambar 2.3 menjelaskan implementasi *load balancing controller* yang digunakan yaitu POX *controller* yang berfungsi untuk mengatur atau mengontrol aplikasi yang ada pada jaringan. Untuk melakukan konfigurasi

komunikasi *openflow* pada POX *controller* menggunakan bahasa pemrograman python. POX *controller* juga tersedia *support modules* untuk *openflow*.



Gambar 2.3 *Controller Software Defined Network*

Sumber : (Kaur, et al., 2014)

2.3 Mininet

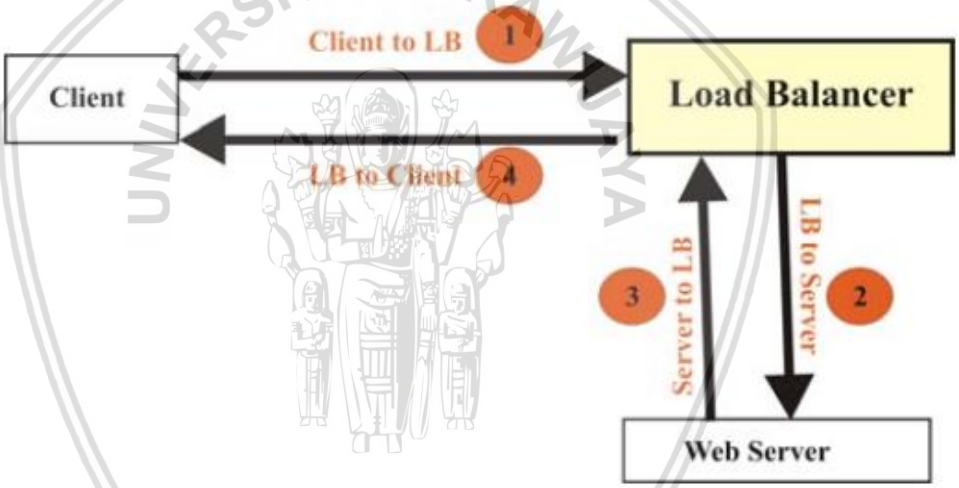
Mininet merupakan aplikasi yang dapat membuat jaringan *host virtual*, *switch*, *controller*, dan link pada satu mesin. *Mininet* bekerja pada satu kernel linux dan memanfaatkan virtualisasi untuk mengemulasikan atau meniru jaringan dengan lengkap hanya memanfaatkan satu sistem saja (Keti & Askar, 2015). *Mininet* sangat penting bagi komunitas *open-source* SDN karena *mininet* biasanya digunakan untuk simulasi, verifikasi, *testing tools*, dan *resource*. *Mininet* juga memiliki API python yang memungkinkan pembuatan dan pengujian jaringan melalui skrip python. Beberapa karakteristik dari terciptanya mininet yaitu:

1. Fleksibilitas, yaitu topologi baru dan fitur baru dapat diataur dalam perangkat lunak menggunakan bahasa pemrograman dan sistem operasi.
2. Penerapan, yaitu implementasi yang benar dilakukan dalam prototipe juga harus digunakan dalam jaringan nyata berdasarkan perangkat keras tanpa perubahan dalam kode sumber.
3. Interaktivitas, yaitu manajemen dan menjalankan simulasi jaringan secara *real time* yang seolah-olah terjadi pada jaringan nyata.
4. Skalabilitas, yaitu *environment prototyping* harus menskalakan jaringan besar dengan ratusan atau ribuan switch hanya pada satu komputer.
5. Realistis, yaitu perilaku prototipe harus mewakili perilaku secara real time dengan tingkat kepercayaan yang tinggi sehingga tumpukan dari aplikasi dan protokol dapat digunakan tanpa modifikasi kode apapun.

6. *Share-able,* yaitu prototipe yang dibuat mudah dibagi dengan kolabolator atau developer lain, yang kemudian dapat menjalankan dan memodifikasi eksperimen yang sudah ada.

2.4 Load Balancing

Load balancing merupakan sebuah teknik untuk mendistribusikan *traffic* yang masuk di antara server yang telah tersedia sehingga *request*/permintaan dapat ditangani dan memberikan respon yang lebih cepat (Rahman, et al., 2014). Tugas dari *load balancing* untuk mendistribusikan beban kerja ke beberapa sumber daya komputasi dan juga untuk mengoptimalkan penggunaan *resource*, memaksimalkan *throughput*, dan menurunkan waktu respons agar tidak membebani sumber daya tunggal (Golinelli, 2015). *Load balancing* juga dapat diimplementasikan pada perangkat lunak maupun perangkat keras tergantung pada jenis beban yang didistribusikan. Perangkat lunak *load balancing* lebih hemat biaya daripada perangkat keras *load balancing* karena tidak memerlukan perangkat keras khusus.



Gambar 2.4 Arsitektur Load Balancing

Sumber : (Kaur & Jyoti, 2017)

Pada Gambar 2.4 menjelaskan tentang arsitektur load balancing. Ketika client ingin mengakses beberapa layanan (service) secara online maka client akan mengirimkan permintaan atau request berupa ip address. Setelah itu permintaan client sampai pada load balancer. Kemudian load balancer mengirimkan permintaan client ke web server yang dipilih sesuai kebijakan tertentu. Web server mengirimkan kembali balasan ke load balancer dan load balancer akan meneruskan balasan tersebut ke client.

Implementasi dari *load balancing* pada *software defined network* memanfaatkan berbagai keunggulan yang ditawarkan. Salah satunya pemisahan antara *control plane (controller)* dan *data plane* untuk melakukan *load balancing* yang efisien, harus dilakukan konfigurasi *software* pada *controller* (Zhong, et al.,

2015). *Load balancing* bekerja dengan memilih *resource* dari daftar *resource* yang sudah tersedia. Ketika menerima *request* kemudian akan di *forward* permintaan tersebut ke *resource* yang dipilih. Ada beberapa metode operasi pada *load balancing.* Misalnya pemilihan *resource* dapat dipilih secara acak dari daftar, tetapi itu mungkin bukan metode yang paling efisien. Contoh lain menggunakan algoritme *Least Connection*, dimana ia memilih *resource* dengan jumlah koneksi yang paling sedikit aktif.

2.4.1 Implementasi Algoritme *Weighted Least Connection*

Algoritme *weighted least connection* hampir sama dengan *least connection*, hanya saja algoritme ini menentukan bobot kinerja dari setiap server (Red Hat, 2014). Setiap server akan diberikan bobot, semakin tinggi nilai bobot yang diberikan pada server maka server akan menerima request lebih banyak dari server yang memiliki nilai bobot rendah. Bobot *default* server adalah 1 dan server yang bebannya 0 tidak akan menerima koneksi aktif apapun. Untuk penentuan bobot dipengaruhi oleh isi *web* atau *web content* yang disediakan oleh *web* server, jika isi *web* bersifat statis *(static web-content)* maka bobot akan dipengaruhi oleh faktor kecepetan media penyimpanan sedangkan jika isi *web* bersifat dinamis *(dynamic web-content)* maka bobot hanya dipengaruhi oleh faktor kecepatan prosesor (Angsar, 2014). Pada sebuah kasus terdapat 4 buah server yang mempunyai jumlah koneksi yang sama, tetapi salah satu server mempunyai bobot yang berbeda yaitu lebih besar dari 3 server lainnya. Jika ada koneksi aktif selanjutnya yang akan masuk, maka sistem akan melakukan perhitungan untuk bobot dari tiap server. Setelah itu koneksi yang baru aktif tersebut akan diteruskan menuju server yang memiliki bobot lebih besar.

Tabel 2.2 *Pseudocode* Algoritme *Weighted Least Connection*

Algoritme *Weighted Least Connection*	
1	Asumsikan terdapat server dengan S = {S0, S1, ..., Sn-1},
2	W(Si) adalah bobot/*weight* dari server Si;
3	C(Si) adalah jumlah koneksi saat ini dari server Si;
4	CSUM = ΣC(Si) (i=0, 1, .. , n-1) adalah jumlah dari nomor koneksi saat
5	ini
6	Koneksi baru diberikan kepada server j dimana (C(Sm) / CSUM)/ W(Sm) =
7	min { (C(Si) / CSUM) / W(Si)} (i=0, 1, . , n-1)
8	Dimana W(Si) tidak bernilai nol
9	Karena CSUM bernilai konstan, tidak perlu lagi dibagi oleh CSUM, maka
10	pada kondisi ini bisa dioptimalkan sebagai berikut
11	C(Sm) / W(Sm) = min { C(Si) / W(Si)} (i=0, 1, . , n-1),
12	Dimana W(Si) tidak bernilai nol
13	Karena operasi pembagian memakan lebih banyak CPU *cycles* daripada
14	operasi perkalian dan linux tidak mengizinkan *mode float* di *kernel*,
15	maka kondisi C(Sm)/W(Sm) > C(Si)/W(Si) dapat dioptimalkan menjadi
16	C(Sm)*W(Si) > C(Si)*W(Sm)
17	
18	for (m = 0; m < n; m++) {
19	if (W(Sm) > 0) {
20	for (i = m+1; i < n; i++) {
21	if (C(Sm)*W(Si) > C(Si)*W(Sm))
22	m = i;
23	}
24	return Sm;
25	}

26	}
27	return NULL;

Sumber: (kb.linuxvirtualserver.org, 2006)

Berdasarkan pada Tabel 2.2, penjelasan dari algoritme *weighted least connection*:

1. Pada baris ke-18 melakukan perulangan (*for)* untuk mengecek seluruh *real* server dalam jumlah n yang dimulai *real* server ke-0.
2. Pada baris ke-19 melakukan seleksi kondisi (*if)* untuk pengecekan bobot server, pada proses sebelumnya dibaris ke-23 variabel awal m=0. Jika bobot atau *weight* lebih dari 0 maka akan dilanjutkan ke proses berikutnya.
3. Pada baris ke-20 melakukan perulangan *(for)* untuk mengecek *real* server berikutnya dalam jumlah n, *real* server dimulai dari ke-1 dari m ditambah 1.
4. Pada baris ke-21 melakukan seleksi kondisi *(if)* untuk pengecekan perbandingan nilai koneksi dan bobot. Jika koneksi dari server sekarang dikalikan bobot server berikutnya (i) lebih besar dengan koneksi server berikutnya dikali bobot server sekarang, maka akan variabel m akan disamakan dengan variabel i pada baris ke-22.

2.5 Web Server

Web server merupakan sebuah *software* atau perangkat lunak yang memberikan layanan berbasis data dan berfungsi sebagai menerima *request* dari HTTP atau HTTPS pada *client* yang biasanya dikenal dengan nama *web browser* dan untuk mengirimkan kembali hasilnya dalam beberapa halaman *web* pada umumnya berbentuk dokumen HTML (Tasneem & Ammar, 2012). Web server biasanya disebut juga sebagai HTTP server, karena basisnya menggunakan protokol HTTP. Cara kerja web server yaitu ketika user mengakses sebuah URL, maka *web browser* akan mengirimkan permintaan ke halaman *web*. Lalu server mencari *web* yang diminta oleh web browser. Setelah itu server akan mengirimkan halaman *web* yang diminta melalui jaringan internet ke *browser* di komputer *client*. Ketika halaman tersebut tiba di komputer, web browser segera menerjemahkan bahasa hypertext atau HTML dan menampilkannya di komputer client, proses ini hanya berjalan dalam hitungan detik. Komputer client dapat mengakses halaman web secara online.

2.6 Httperf

Httperf merupakan *tools* untuk mengukur kinerja *web* server melalui protokol HTTP. Protokol ini support HTTP/1.0 dan HTTP/1.1 yang digabungkan dengan berbagai generator beban kerja (Golinelli, 2015). *Httperf* berfungsi dengan meniru sejumlah *client* yang mengakses situs web tertentu, lalu menginduksi beban di server, karena semua *request* dijalankan dari program yang sama maka dapat mengukur respon server. Pada penelitian ini *httperf* digunakan untuk melakukan pengujian terhadap *web* server dengan menggunakan beberapa parameter

tertentu. Parameter yang digunakan untuk pengujian ini yaitu *connection rate, throughput,* dan CPU *usage*.

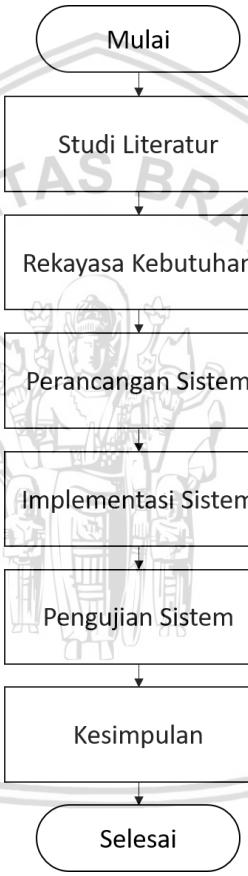
1. *Connection rate* merupakan banyaknya jumlah koneksi yang dilakukan oleh *client* dalam satu waktu (*rate*). Semakin besar nilai *connection rate* maka semakin baik kualitas jaringan tersebut, tetapi jika semakin kecil nilai dari *connection rate* maka semakin buruk kualitas jaringan tersebut.
2. *Response time* merupakan waktu tanggap yang diberikan ketika user melakukan request ke server. Semakin kecil nilai *response time* maka semakin baik kualitas jaringan tersebut, tetapi jika semakin besar nilai *response time* maka semakin buruk kualitas jaringan tersebut.
3. *Throughput* merupakan ukuran data yang dikirim dibagi dengan satu satuan waktu. Semakin besar nilai *throughput* maka semakin baik kualitas jaringan tersebut, tetapi jika semakin kecil nilai dari *throughput* maka semakin buruk kualitas jaringan tersebut.
4. CPU *usage* merupakan persentase jumlah penggunaan resource yang diperlukan program untuk menjalankan instruksi-instruksinya. Semakin kecil nilai cpu *usage* maka semakin baik kualitas sebuah sistem, sedangkan semakin besar nilai cpu *usage* maka semakin burun kualitas sebuah sistem.

2.7 *Library* POX

POX adalah salah satu *controller software defined network* yang digunakan untuk memudahkan pengguna untuk melakukan konfigurasi pada jaringan *software defined network*. POX juga menyediakan modul khusus untuk komunikasi *OpenFlow* dan metode-metode API untuk berinteraksi dengan *Openflow switch*. Komponen yang ditambahkan akan menghubungkan API yang telah tersedia termasuk *routing*, topologi (LLDP), *host tracking*, dan python *interface* yang diimplementasikan sebagai pembungkus untuk komponen API.

BAB 3 METODOLOGI

Pada bab ini menjelaskan studi literatur, rekayasa kebutuhan, perancangan sistem, implementasi sistem, pengujian system, dan kesimpulan untuk penelitian yang berjudul Implementasi *Load Balancing* Di *Web* Server Dengan Algoritme *Weighted Least Connection* Pada *Software Defined Network*. Penelitian ini bersifat implementatif pengembangan berupa implementasi *load balancing* menggunakan algoritme *weighted least connection* pada jaringan *software defined network.* Selain itu pengambilan kesimpulan dari hasil pengujian sistem untuk digunakan kedepannya sebagai pengembangan penelitian yang lebih lanjut. Tahapan-tahapan yang digunakan untuk penelitian ini dapat dilihat pada Gambar 3.1.



Gambar 3.1 Diagram Alir Metode Penelitian

3.1 Studi Literatur

Pada sub bab ini menjelaskan studi literatur yang dilakukan berdasarkan pemahaman dasar-dasar teori yang terkait dengan beberapa konsep sebagai berikut.

1. *Software Defined Network*
2. *Controller*
3. *Openflow*

4. *Mininet*
5. Algoritme *Weighted Least Connection*
6. *Web* Server
7. Parameter Pengujian
8. *Load Balancing*

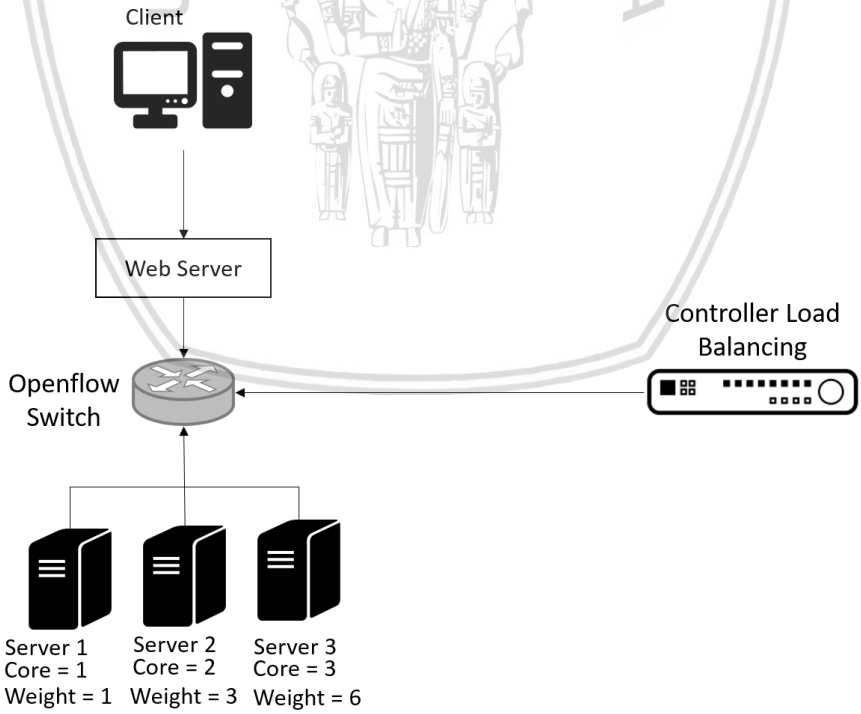
Literatur yang digunakan dalam penelitian ini didapatkan dari berbagai sumber seperti buku, *paper* (jurnal ilmiah), dan dokumen dari *internet* yang memiliki relevansi dengan penelitian ini.

3.2 Rekayasa Kebutuhan

Pada sub bab ini menjelaskan rekayasa kebutuhan untuk menentukan kebutuhan apa saja yang diperlukan. Kebutuhan sistem dapat mempermudah proses perancangan dan implementasi. Kebutuhan pada penelitian ini terdiri dari kebutuhan fungsional dan kebutuhan non-fungsional.

3.3 Perancangan Sistem

Perancangan sistem dilakukan setelah seluruh kebutuhan sistem dilakukan agar pembangunan sistem pada penelitian ini dapat terarah dan terstruktur. Perancangan sistem dapat dilihat pada Gambar 3.2.



Gambar 3.2 Perancangan Sistem

Pada Gambar 3.2 merupakan perancangan sistem yang digunakan pada topologi *mininet* server dengan menggunakan tiga server *virtual* yang masing-masing akan berkomunikasi pada *port* 80. Pada ketiga server ini memiliki

spesifikasi yang berbeda, diantaranya server 1 memilki jumlah *core* sebanyak 1, server 2 memiliki jumlah *core* sebanyak 2, dan server 3 memilki jumlah *core* sebanyak 3. Pada masing-masing server *virtual* akan diberikan bobot, server 1 akan diberikan bobot sebanyak 1, server 2 sebanyak 3, dan server 3 sebanyak 6. Server yang memiliki nilai bobot yang besar akan menampung jumlah koneksi yang banyak dibanding server lainnya. Setelah itu *openflow switch* bertugas meneruskan paket ke *controller* untuk diberikan *action* terhadap paket tersebut, lalu *traffic* paket akan dibagi berdasarkan algoritme *weighted least connection*. Kemudian dilakukan pengujian performansi dan kinerja dari *web* server dengan melakukan *request* pada *client*. Perancangan sistem ini terdiri dari tiga *host* sebagai server, satu *host* sebagai *client*, satu buah *switch*, dan satu buah *controller*.

3.3.1 Perancangan Algoritme

Perancangan algoritme dilakukan setelah perancangan sistem selesai. Algoritme *load balancing* yang digunakan yaitu *weighted least connection* yang dimana algoritme ini bekerja dengan menentukan bobot kinerja dari setiap server. Algoritme ini akan dijalankan menggunakan POX *controller*.

3.4 Implementasi Sistem

Implementasi sistem dilakukan setelah perancangan sistem dan perancangan algoritme selesai. Implementasi dilakukan sesuai dengan perancangan sistem. Pertama melakukan instalasi *mininet* pada sistem operasi linux, lalu melakukan instalasi *controller* POX, setelah itu membangun arsitektur *load balancing web* server *pada software defined network* di *mininet* server, kemudian membuat algoritme *weighted least connection* sebagai mekanisme *load balancing*, selanjutnya melakukan akses ke halaman *web* server, dan yang terakhir implementasi pengujian dengan parameter *connection rate*, *response time, throughput,* dan CPU *usage* untuk mengetahui performa dan kinerja *web* server.

3.5 Pengujian Sistem

Pengujian dilakukan setelah mengetahui kinerja sistem apakah kebutuhan yang sudah di analisis sesuai atau tidak. Pengujian pada penelitian ini akan mengacu pada beberapa parameter yaitu jumlah koneksi yang ditampung setiap server, *connection rate*, *response time throughput,* dan CPU *usage*. Sehingga dapat disusun beberapa pengujian antara lain:

1. Melakukan *request* menggunakan 1 *client* dengan memberikan jumlah *request* pada server sebanyak 3000 *request* dan *rate* 300, 150, dan 75 *conn/sec.*
2. Melakukan *request* menggunakan 1 *client* dengan memberikan jumlah *request* pada server sebanyak 500 *request* dan *rate* 50, 25, dan 13 *conn/sec.*
3. Memberikan nilai bobot (*weight*) pada tiap server secara berurutan yaitu 1,3,6.

3.6 Kesimpulan

Kesimpulan dilakukan setelah semua tahapan telah selesai dilakukan, yaitu dari tahap analisis kebutuhan hingga tahap pengujian dan analisis sistem. Kesimpulan diambil berupa jawaban atas rumusan masalah yang telah ditentukan sebelumnya. Saran untuk memberikan pertimbangan dengan mengembangkan penelitian yang lebih lanjut.

BAB 4 REKAYASA KEBUTUHAN

4.1 Deskripsi Umum Sistem

Sistem *load balancing* menggunakan algoritme *weighted least connection* merupakan sistem penyeimbang beban trafik untuk layanan *web* dengan jumlah koneksi dan nilai bobot (*weight*) sebagai faktor pembagian beban. Sistem ini menggunakan *flow* sebagai penentu jumlah koneksi pada server. Untuk menerapkan sistem ini dibutuhkan *controller* yang berfungsi untuk mengelola *flow* ke *switch* menggunakan protokol *openflow*. Lalu *flow* akan disimpan dalam *flow table* yang berada pada perangkat *switch*. Untuk pembagian beban pada server, informasi dari *flow* tersebut akan dikelola menggunakan algoritme *weighted least connection* yang sudah diterapkan pada *controller* tersebut.

Sistem *load balancing* ini diterapkan pada arsitektur *software defined network*, sehingga dibutuhkan sebuah lingkungan sistem yang mendukung arsitektur tersebut. Oleh karena itu, penelitian ini menggunakan *mininet* sebagai simulator. Karena sistem *load balancing* ini akan diterapkan pada arsitektur *software defined network*, maka dibutuhkan subsistem yang saling bekerja sama untuk membentuk sistem *load balancing* yang utuh. Subsistem tersebut terdiri dari subsistem *controller*, subsistem *switch*, subsistem *web* server, dan subsistem *client*. Berikut ini penjelasan dari masing-masing subsistem:

- Subsistem *Controller*, pada subsistem ini merupakan inti dari sebuah arsitektur jaringan *software defined network* yang bertugas untuk mengatur atau memutuskan aksi tertentu pada paket yang masuk ke jaringan. Subsistem ini juga akan diterapkan metode *load balancing* menggunakan algoritme *weighted least connection*.
- Subsistem *Switch*, pada subsistem ini berfungsi untuk menerima paket yang masuk ke jaringan dan melakukan aksi terhadap paket tersebut berdasarkan informasi yang tersimpan pada *flow table*. *Flow table* merupakan informasi mengenai paket yang masuk ke jaringan dan tersimpan di dalam *switch*. Informasi tersebut akan digunakan untuk menentukan apa yang harus dilakukan terhadap paket tersebut.
- Subsistem *Web* Server, pada subsistem ini berfungsi untuk menampung *request* dengan cara bekerja secara bergantian melayani *request* yang dilakukan oleh *client*. Subsistem ini nantinya juga akan diberikan nilai bobot (*weight*) sesuai spesifikasinya.
- Subsistem *Client*, pada subsistem ini hanya digunakan pada saat pengujian sistem *load balancing*. Subsistem ini bertugas untuk mengirimkan *request* terhadap *web* server.

4.2 Analisis Kebutuhan

Analisis kebutuhan sistem bertujuan untuk mengetahui kebutuhan apa saja yang diperlukan dalam membangun sistem ini. Kebutuhan sistem dapat

mempermudah proses perancangan dan implementasi. Kebutuhan penelitian ini terbagi menjadi dua yaitu kebutuhan fungsional dan kebutuhan non-fungsional.

4.2.1 Kebutuhan Fungsional

Kebutuhan fungsional merupakan kebutuhan yang berisi tentang proses-proses yang dilakukan oleh sistem dan informasi apa saja yang harus ada dan dihasilkan oleh sistem. Kebutuhan fungsional pada sistem ini akan dibagi menjadi beberapa subsistem agar lebih memudahkan proses perancangan diantaranya sebagai berikut.

4.2.1.1 Subsistem *Controller*

- *Controller* dapat menjalankan modul *load balancing* menggunakan POX.
- *Controller* mampu menerima paket dari *switch*.
- *Controller* dapat menentukan atau mengarahkan paket ke *web* server dengan jumlah koneksi dan nilai bobot (*weight*).

4.2.1.2 Subsistem *Switch*

- *Switch* dapat menerima paket dari *client*.
- *Switch* dapat mengirimkan pesan informasi paket yang akan datang kepada *controller* (*Packet In Message*).
- *Switch* dapat menerima pesan informasi mengenai tindakan yang harus dilakukan terhadap paket, dimana pesan tersebut dikirimkan oleh *controller* (*Packet Out Message*).

4.2.1.3 Subsistem *Web* Server

- *Web* server dapat memberikan respon terhadap *request* dari *client*.
- *Web* server akan diberikan nilai bobot (*weight*) sesuai dengan spesifikasi *web* server.

4.2.1.4 Subsistem *Client*

- *Client* dapat mengirimkan *request* terhadap *web* server.

4.2.2 Kebutuhan Non-Fungsional

Kebutuhan non-fungsional merupakan kebutuhan yang harus ada pada sistem. Kebutuhan non-fungsional pada sistem ini terdiri dari kebutuhan perangkat lunak dan kebutuhan perangkat keras.

4.2.2.1 Kebutuhan Perangkat Lunak

Berikut ini merupakan beberapa perangkat lunak yang harus ada dalam sistem yaitu:

- Menggunakan sistem operasi Linux Ubuntu 14.04.
- Menggunakan *mininet* versi 2.2.2 sebagai simulator jaringan.
- Menggunakan *openflow* versi 1.3 sebagai protokol yang digunakan.
- Menggunakan *httperf* untuk mengetahui atau menguji performa dari sistem *load balancing* terhadap *web* server.

4.2.2.2 Kebutuhan Perangkat Keras

Berikut ini merupakan beberapa perangkat keras yang harus ada dalam sistem yaitu:

- Menggunakan *processor* Intel *core* i5-3230M CPU @2.60 GHz.
- Menggunakan RAM 4 GB.
- *Harddisk* kapasitas 1000 GB.

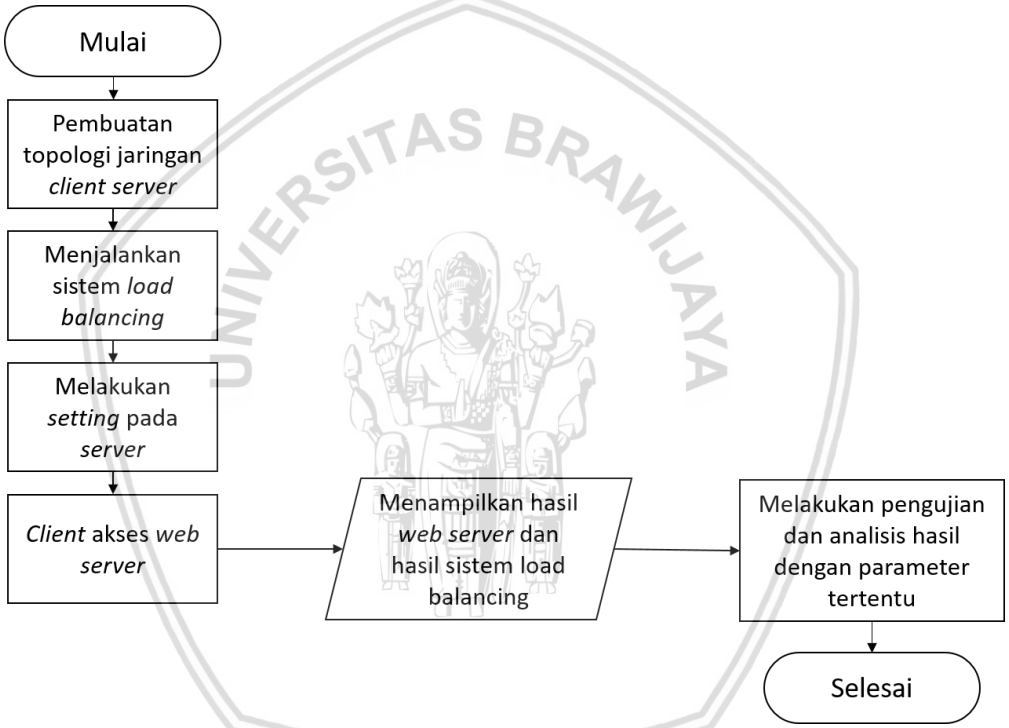
BAB 5 PERANCANGAN DAN IMPLEMENTASI

Pada bab ini menjelaskan tentang tahap-tahap perancangan yang dilakukan dalam pengerjaan penelitian ini. Bagian ini juga menjelaskan lebih mendalam pada konsep perancangan *load balancing*, alur komunikasi, pengecekan paket, alur sistem, dan diagram blok perancangan sistem.

5.1 Perancangan

5.1.1 Perancangan Arsitektur *Software Defined Network*

Pada sub bab ini menjelaskan perancangan sistem dimulai dengan tahapan yang ada pada bab sebelumnya hingga pengujian yang dilakukan terhadap sistem.



Gambar 5.1 Diagram Alir Perancangan Arsitektur *Software Defined Network*

Pada Gambar 5.1 menjelaskan perancangan arsitektur SDN yang dilakukan, pada tahap pertama dimulai dengan pembuatan topologi jaringan *client* server pada *mininet* yang nantinya digunakan untuk melakukan implementasi *load balancing*. Pada tahap ini terdapat empat *host* yang masing-masing tiga *host* sebagai server dan satu *host* sebagai *client*. Tahap selanjutnya menjalankan sistem *load balancing* dan melakukan *setting* pada server. Setelah itu dilanjutkan melakukan *setting* pada *client* yang akan digunakan untuk melakukan pengujian. Lalu *client* melakukan komunikasi dengan mengakses *web* server yang sudah di*setting* pada tahap sebelumnya. Setelah permintaan *client* kepada *web* server berjalan, akan ditampilkan hasil dari sistem *load balancing* menggunakan *algoritme weighted least connection*. Tahap terakhir dari perancangan sistem ini yaitu melakukan pengujian terhadap sistem dengan melakukan analisis hasil

pengujian dengan parameter yang sudah ditentukan seperti *connection rate*, *response time, throughput,* dan CPU *usage*.

5.1.1.1 *POX* Controller

POX controller merupakan sebuah *controller software defined network* yang ditulis dengan bahasa pemrograman python versi 2.7. POX meyediakan modul khusus untuk komunikasi pada protokol *OpenFlow* versi 1.3. Pada penelitian ini sistem melakukan *load balancing* akan dibuat menggunakan *library* POX. *Load balancing* ini akan melakukan instruksi pada *switch* untuk meneruskan paket dan pada saat bersamaan juga beban *traffic* akan menyeimbangkan menggunakan algoritme *weighted least connection*.

5.1.1.2 Open vSwitch

Open vSwitch merupakan aplikasi yang bertindak sebagai *OpenFlow switch* pada arsitektur *software defined network*. Aplikasi ini juga dapat digunakan sebagai *virtual switch* pada *mininet*.

5.1.1.3 Web Server

Pada penelitian ini *web* server yang digunakan adalah modul *web* server pada python yaitu SimpleHTTPServer. Modul tersebut sudah terdapat pada modul standar python.

5.1.1.4 *Alokasi Pengalamatan IP*

Komponen	Alamat IP
Web Server 1	10.0.0.1
Web Server 2	10.0.0.2
Web Server 3	10.0.0.3
Client 1	10.0.0.4
Client 2	10.0.0.5
Client 3	10.0.0.6
Client 4	10.0.0.7

5.1.2 Perancangan Algoritme *Weighted Least Connection*

Algoritme *weighted least connection* dipasang pada modul *load balancing* yang nantinya akan dijalankan menggunakan POX *Controller*. Berikut *pseudocode* dari algoritme *weighted least connection*.

Tabel 5.1 *Pseudocode* Perancangan Algoritme *Weighted Least Connection*

Perancangan Algoritme *Weighted Least Connection*	
1	menerima informasi koneksi dari flow
2	simpan semua server yang aktif ke variabel serverSet
3	inisialisasi variabel selected_server adalah server pertama

```
4   for (semua server pada server set)
5    if (server pertama bisa melayani koneksi)
6     server selanjutnya = index server pertama + 1
7     for (server selanjutnya sampai seluruh serverSet)
8      if (koneksi dari server sekarang * bobot server berikutnya > koneksi
9   server berikutnya * bobot server sekarang)
10        index terbesar = index server selanjutnya
11    selected_server = server dengan index terbesar
```

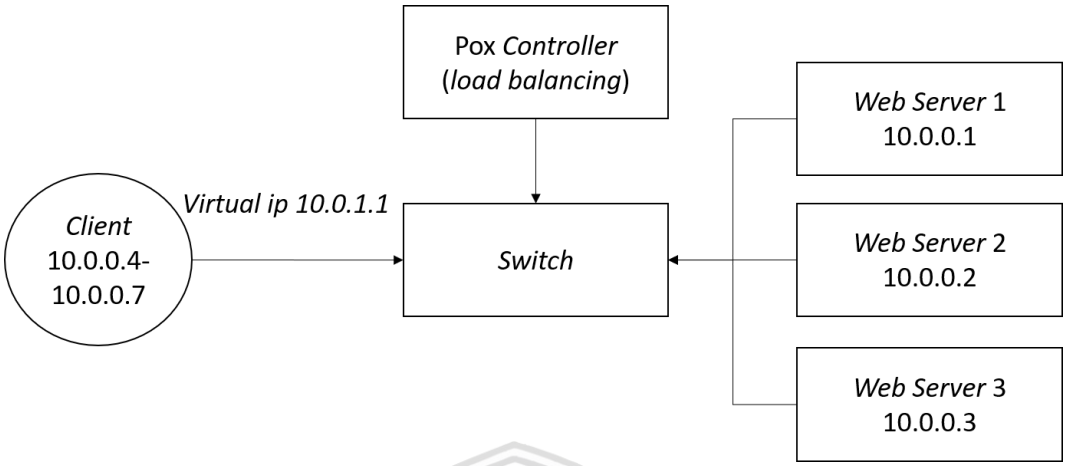
Berdasarkan pada Tabel 5.1, penjelasan dari perancangan algoritme *weighted least connection*:

1. Pada baris pertama menjelaskan tahapan menerima informasi jumlah koneksi dari *flow*.
2. Pada baris ke-2 menjelaskan semua server yang aktif akan disimpan pada variabel serverSet.
3. Pada baris ke-3 menjelaskan server pertama diinisialisasi pada variabel selected_server.
4. Pada baris ke-4 melakukan perulangan *(for)* untuk mengecek seluruh server yang aktif pada variabel serverSet.
5. Pada baris ke-5 melakukan seleksi kondisi *(if)* untuk pengecekan apakah server pertama masih bisa melayani koneksi.
6. Pada baris ke-6 menjelaskan jika server pertama masih bisa melayani koneksi maka server selanjutnya = *index* server pertama + 1.
7. Pada baris ke-7 melakukan perulangan *(for)* untuk mengecek seluruh server dari *index* berikutnya.
8. Pada baris ke-8 sampai ke-9 melakukan seleksi kondisi *(if)* untuk melakukan perbandingan nilai koneksi dan bobot.
9. Pada baris ke-10 menjelaskan *index* terbesar yaitu *index* server selanjutnya jika memiliki nilai bobot yang lebih besar dan jumlah koneksi aktif paling sedikit.
10. Pada baris ke-11 menjelaskan koneksi yang nantinya akan masuk atau server yang dipilih adalah server dengan *index* terbesar.

Untuk mengetahui berapa jumlah koneksi yang masuk pada server yang aktif atau tersedia dapat dihitung dengan perhitungan sebagai berikut:

$$\frac{Nilai\ Weight}{Jumlah\ Weight}\ x\ Total\ Koneksi\ Yang\ Diberikan = Jumlah\ Koneksi\ Yang\ Masuk$$

5.1.3 Gambaran Komunikasi



Gambar 5.2 Gambaran Komunikasi

Pada Gambar 5.2 menjelaskan alur komunikasi yang akan dilakukan oleh *client.* Pertama *client* akan mengakses ip *virtual web* server dengan alamat 10.0.1.1. Kemudian *client* akan terhubung ke *switch* yang bertugas untuk meneruskan permintaan dari *client* menuju alamat *destination*. *Switch* akan terhubung dengan sebuah *controller* yang bertugas untuk membuat keputusan terhadap paket yang masuk. Selanjutnya sistem *load balancing* akan membagi *request* atau permintaan dari *client* berdasarkan jumlah server yang tersedia. *Request* dari *client* akan didistribusikan ke server berdasarkan algoritme *weighted least connection* Untuk koneksi pertama kali masuk akan diteruskan ke server yang pertama.

Algoritma *weighted least connection* akan memilih server yang memiliki jumlah koneksi paling sedikit dan nilai bobot yang besar, maka *request* dari *client* akan diteruskan ke server tersebut. Setiap *client* melakukan *request* kepada *web* server maka *controller* akan memberikan keputusan dalam pembagian beban server sehingga server tidak mengalami *overload* karena banyaknya *request* dari *client*. Masing-masing server memiliki data yang sama yaitu halaman *html web* server yang akan diakses oleh *client*. Sehingga respon yang diberikan kepada *web* server pun juga sama. Alamat *host client* pada ip 10.0.0.4 sampai 10.0.0.7 sedangkan alamat *host* server pada ip 10.0.0.1 sampai 10.0.0.3.

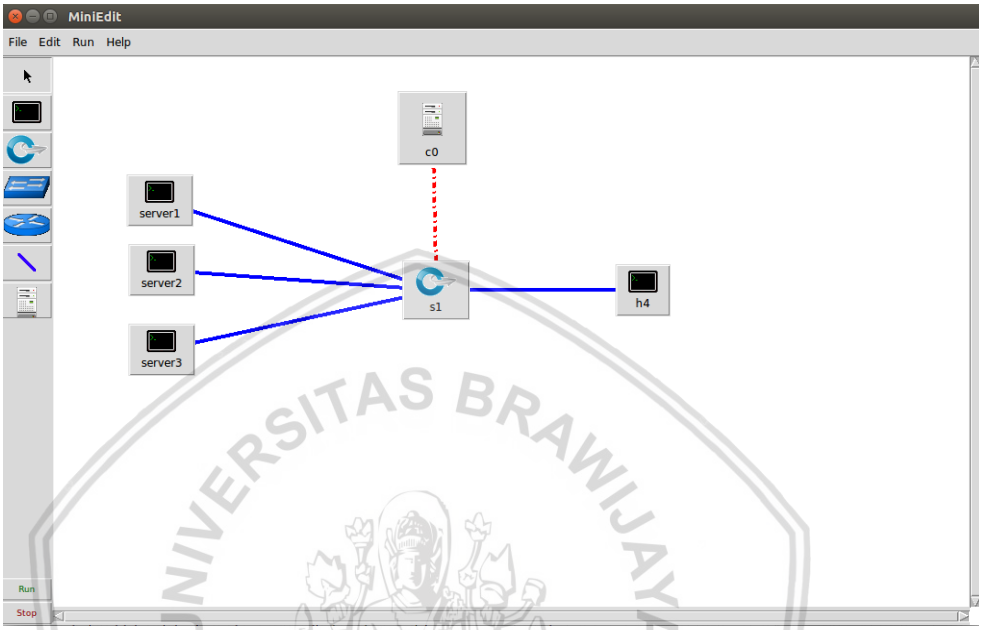
5.2 Implementasi

5.2.1 Membuat Topologi

Untuk melakukan implementasi *load balancing* di *web* server dengan algoritme *weighted least connection* pada *software defined network* membuat topologi menggunakan *mininet* dengan *single switch,* empat *host* dan *controller* pox sebagai *remote controller*. Berikut command untuk pembuatan topologi pada *mininet*:

```
1   $ sudo mn ~/mininet/examples/miniedit.py
```

Pada *command* diatas menjelaskan pembuatan topologi ini menggunakan miniedit yang merupakan mininet berbasis GUI. Lalu akan muncul tampilan miniedit seperti pada Gambar 5.3. Setelah itu membuat topologi dengan *single switch* dan jumlah *host* sebanyak empat, kemudian mengatur alamat *MAC address* secara otomatis dan arp antar *host*, dan juga mengatur *remote controller* yang akan digunakan sebagai pengatur *traffic* paket.



Gambar 5.3 Pembuatan Topologi

Pada Gambar 5.3 merupakan hasil dari pembuatan topologi. Dari gambar diatas dapat diketahui bahwa menambahkan *host* sebanyak empat (server1, server2, server3, h4), *single switch* (s1), dan *controller* (c0). Untuk menghubungkan masing-masing *host* dilakukan *adding links* dari (server1,s1) sampai (h4,s1) yang berarti server1 sampai h4 terhubung dengan satu *switch*. Setelah selesai dalam pembuatan topologi, selanjutnya *controller* dan *switch* akan menjalankan sistem.

5.2.2 Menjalankan *Load Balancing*

Command Untuk menjalankan sistem *load balancing* dengan algoritma *weighted least connection* pada *software defined network* menggunakan pox *controller* sebagai berikut:

```
1   $ ./pox.py log.level --DEBUG misc.ip_loadbalancer --
    ip=10.0.1.1 --servers=10.0.0.1,10.0.0.3,10.0.0.2
```

Pada *command* diatas menjelaskan bahwa pox.py merupakan *script* utama pada POX untuk menjalankan *script* yang telah dibuat yaitu ip_loadbalancer.py. Lalu terdapat *command* log.level yaitu untuk menampilkan pesan-pesan pada *mode debug*. Setelah itu mengatur alamat ip virtual yang akan digunakan oleh *client* untuk mengakses *web* server dengan ip 10.0.1.1 dan server= 10.0.0.1,10.0.0.3,10.0.0.2 merupakan alamat IP *web* server.

```
hafiskarim@hafiskarim-Satellite-C40-A: ~/pox
hafiskarim@hafiskarim-Satellite-C40-A:~$ cd pox
hafiskarim@hafiskarim-Satellite-C40-A:~/pox$ ./pox.py log.level --DEBUG misc.ip_
loadbalancer --ip=10.0.1.1 --servers=10.0.0.1,10.0.0.3,10.0.0.2
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
DEBUG:core:POX 0.2.0 (carp) going up...
DEBUG:core:Running on CPython (2.7.6/Nov 23 2017 15:49:48)
DEBUG:core:Platform is Linux-4.4.0-104-generic-x86_64-with-Ubuntu-14.04-trusty
INFO:core:POX 0.2.0 (carp) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[None 1] closed
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
INFO:iplb:IP Load Balancer Ready.
INFO:iplb:Load Balancing on [00-00-00-00-00-01 2]
INFO:iplb.00-00-00-00-00-01:Server 10.0.0.1 up
INFO:iplb.00-00-00-00-00-01:Server 10.0.0.3 up
INFO:iplb.00-00-00-00-00-01:Server 10.0.0.2 up
```

Gambar 5.4 Menjalankan *Load Balancing*

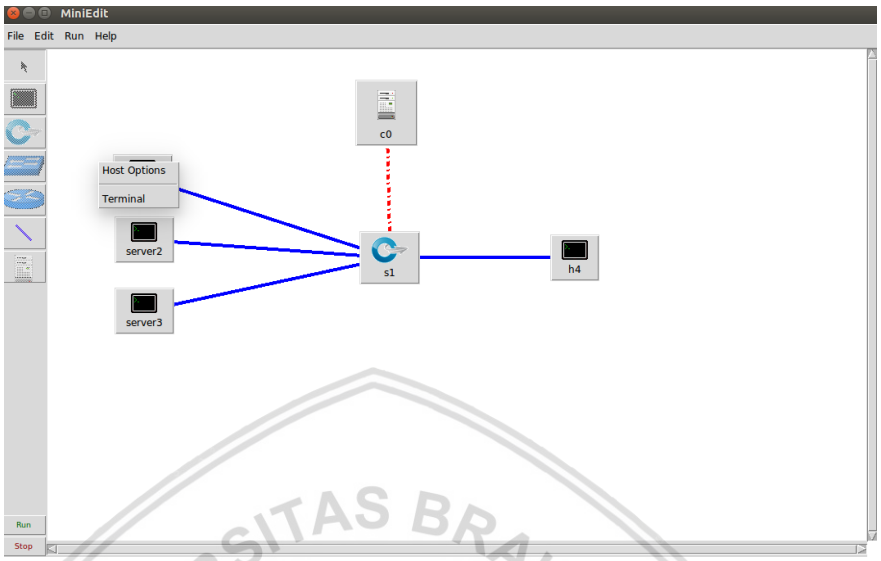
Pada Gambar 5.4 merupakan hasil dari *command* untuk *menjalankan load balancing.* Dari gambar diatas dapat diketahui ketika *controller* pox sudah aktif maka *controller* akan melakukan *listening* pada *port* 6633, maka sistem *load balancing* siap untuk dijalankan. Lalu server dengan ip 10.0.0.1 sampai 10.0.0.3 akan *up* atau aktif.

5.2.2.1 *Setting* Server Heterogen

Untuk melakukan *setting* server heterogen menggunakan *mininet* dengan mengisi core/jumlah prosesor pada tiap server klik properties pada setiap host, lalu mengisi core seperti pada Gambar 5.5 yang menunjukkan pengesetan *core*/jumlah prosesor pada setiap server. Pada server 1 di set dengan jumlah prosesor 1, server 2 di set dengan jumlah prosesor 2, dan server 3 di set dengan jumlah prosesor 3.

Gambar 5.5 Pengesetan *Core*/Jumlah Prosesor Pada Server **1, 2, dan 3**

Pada Gambar 5.6 menjelaskan untuk membuka terminal pada setiap *host* dengan klik kanan pada *host* yang dipilih lalu klik terminal. Pada host ini nantinya berfungsi sebagai server untuk menerima *request* dari *client*.

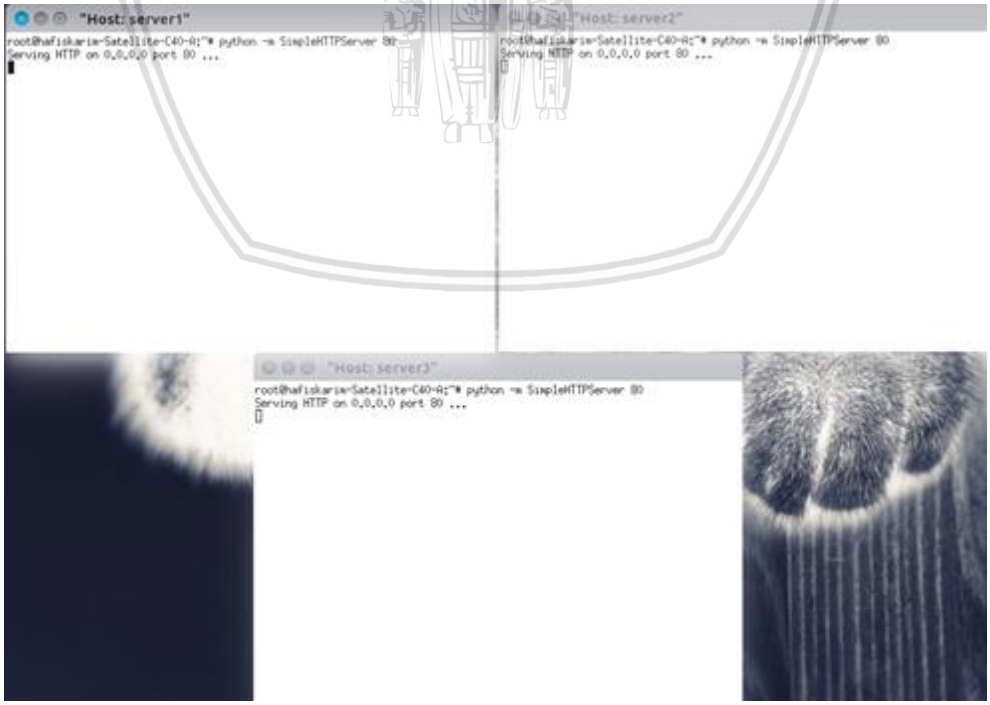


Gambar 5.6 Menjalankan Server

Setelah membuka terminal dari ketiga *host* tersebut, melakukan *command* sebagai berikut:

```
1   #python -m SimpleHTTPServer 80
```

Pada *command* diatas menjelaskan untuk setting server dengan menggunakan protokol HTTP pada *port* 80.



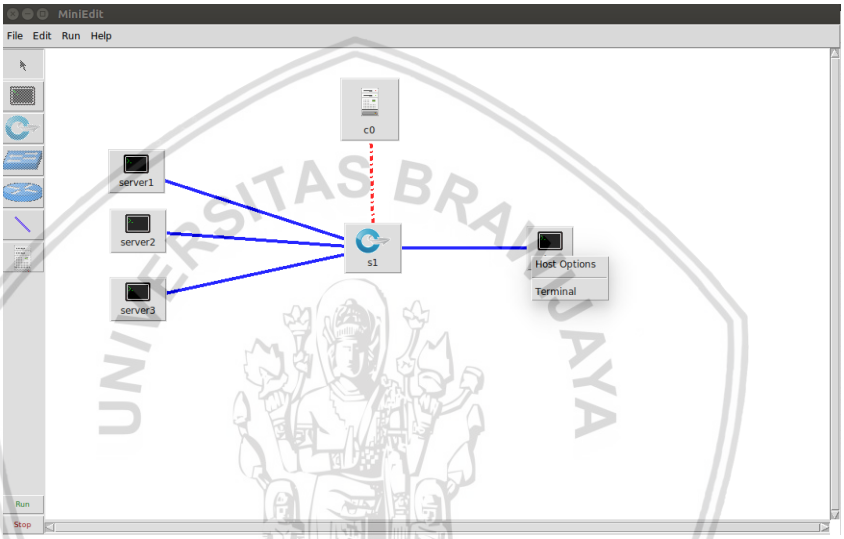
Gambar 5.7 *Setting* Server

Pada Gambar 5.7 merupakan hasil dari *client* melalukan *request* ke *web* server. Dari gambar diatas dapat diketahui bahwa *request client* mengunakan *command #curl* telah dikirim oleh server yang berupa *doctype html.*

5.2.2.2 Setting Client

Untuk melakukan pengujian *load balancing* pada *web* server menggunakan *miniedit* diperlukan *client* untuk melakukan *request*. Berikut gambar untuk melakukan *setting client* pada *host* menggunakan *miniedit*.

Pada Gambar 5.8 menjelaskan untuk membuka terminal pada setiap *host* dengan klik kanan pada *host* yang dipilih lalu klik terminal. Pada *host* ini nantinya digunakan *client* untuk melakukan *request*.



Gambar 5.8 *Setting Client*

Gambar 5.9 Menjalankan *Client*

Pada Gambar 5.9 merupakan hasil dari *setting client* pada sebuah *host*. Dari gambar diatas adalah tampilan terminal dari *host* 4 yang nantinya berfungsi sebagai *client* untuk melakukan *request* ke *web* server.

5.2.2.3 Client *Akses* Web Server

Untuk melakukan implementasi *load balancing* dari sisi *client* dapat menuliskan *command #curl* pada alamat IP 10.0.1.1. Lalu akan muncul halaman *doctype html* pada *web* server yang nantinya data tersebut digunakan untuk melakukan *request* ke server.

Gambar 5.10 *Client Akses Web* Server

Pada Gambar 5.10 merupakan hasil dari *client* melalukan *request* ke *web* server. Dari gambar diatas dapat diketahui bahwa *request client* mengunakan *command #curl* telah dikirim oleh server yang berupa *doctype html.*

5.2.3 Sistem *Load Balancing Weighted Least Connection*

Sistem *load balancing* pada arsitektur *software defined network* untuk melakukan pemilihan server yang dilakukan oleh *controller*. Berikut kode sumber sistem *load balancing* dengan algoritme *weighted least connection*:

Tabel 5.2 Kode Sumber Algoritme *Weighted Least Connection*

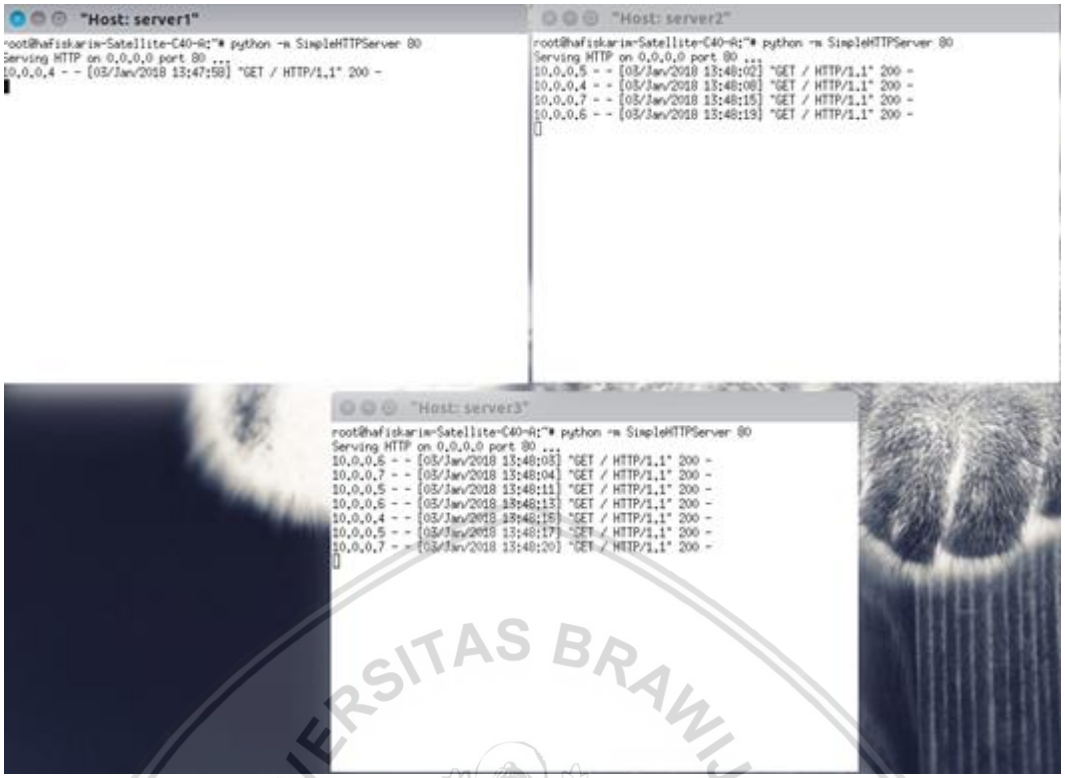
Algoritme *Weighted Least Connection*

```
1   selected_server=0
2   resourceUsedPercentage=[0,0,0]
3   resourceServerPercentage=[1,3,6]
4   connectionWeight = 0.01
5   def _pick_server(self,key, inport):
6      global selected_server, resourceUsedPercentage,
7   resourceServerPercentage, connectionWeight
8      serverSet=self.live_servers.keys()
9      serverCount=len(serverSet)
10     m=0
11     for m in range (m, (serverCount)):
12          if(resourceServerPercentage[m]>0):
13              i=m+1
14              for i in range (i, (serverCount)):
15  if(resourceUsedPercentage[m]*resourceServerPercentage[i]>resourceUsedP
16  ercentage[i]*resourceServerPercentage[m]):
17                      m=i
18              selected_server=serverSet[m]
19              resourceServerPercentage[m]-=connectionWeight
20              resourceUsedPercentage[m]+=connectionWeight
21              return selected_server
22     return None
```

Berdasarkan pada Tabel 5.2 dapat diketahui kode sumber algoritme *weighted least connection* pada *controller* pox dengan bahasa python. Berikut penjelasan dari tabel diatas.

- Pada baris ke-1 melakukan inisialisasi variabel global selected_server untuk pemilihan server yang dimulai dari 0.
- Pada baris ke-2 melakukan inisialisasi variabel global resourceUsedPercentage merupakan persentase *resource* yang digunakan perserver dan untuk menyimpan banyaknya *resource* yang digunakan.
- Pada baris ke-3 melakukan inisialisasi variabel global resourceServerPercentage untuk perbandingan weight/bobot pada server, yang dimana perbandingan server 1 dengan server 2 merupakan 3 kali lipatnya dari server 1 dan server 3 merupakan 6 kali lipatnya dari server 1.
- Pada baris ke-4 inisialisasi variabel global connectionWeight merupakan bobot resource yang dipakai untuk 1 koneksi yaitu sebanyak 1%.
- Pada baris ke-6 sampai ke-7 melakukan pengambilan nilai dari variabel global untuk selected_server, resourceUsedPercentage, resourceServerPercentage, dan connectionWeight.
- Pada baris ke-8 menjelaskan daftar server aktif yang berisikan ip.
- Pada baris ke-9 menjelaskan jumlah server yang aktif didapat dari *array* serverSet.
- Pada baris ke-10 menjelaskan variabel m untuk *indexing* server yang dimulai dari 0.

- Pada baris ke-11 melakukan perulangan *for* untuk pengecekan seluruh server.
- Pada baris ke-12 melakukan seleksi kondisi *if* untuk pengecekan server apakah server masih mampu untuk menangani koneksi.
- Pada baris ke-13 menjelaskan variabel i untuk *indexing* server berikutnya dengan ditambah m+1.
- Pada baris ke-14 melakukan perulangan *for* untuk mengecek seluruh server dari *index* berikutnya.
- Pada baris ke-15 sampai ke-16 merupakan algoritme *weighted least connection* dengan melakukan seleksi kondisi *if* untuk mengecek perbandingan nilai koneksi dan bobot.
- Pada baris ke-17 melakukan inisialisasi server yang sebelumnya memiliki bobot yang lebih besar dan koneksi yang aktif paling sedikit.
- Pada baris ke-18 menjelaskan koneksi yang masuk akan dimasukkan ke server dengan *index* m.
- Pada baris ke-19 menjelaskan pengurangan resource untuk meyimpan sisa resource pada server yang dipilih akan berkurang 1% karena adanya koneksi yang masuk dari *client*.
- Pada baris ke-20 menjelaskan penambahan resource untuk menyimpan banyaknya resource yang digunakan pada server yang dipilih akan bertambah 1%.
- Pada baris ke-21 untuk mengembalikan ip server mana yang akan dipilih.
- Pada baris ke-22 menjelaskan jika bobot semua server tidak bisa dimasukkan koneksi baru.



Gambar 5.11 *Weighted Least Connection Directed* Server

Pada Gambar 5.11 merupakan hasil setelah menjalankan program *load balancing*, yang dimana membandingkan jumlah koneksi dan nilai bobot pada masing-masing server. Server yang memiliki jumlah koneksi aktif paling sedikit dan nilai bobot yang besar maka server tersebut akan menampung koneksi baru dari *client*.

BAB 6 PENGUJIAN

Pada bab ini dilakukan pengujian untuk mengetahui performa beserta hasil pengujian dari penerapan *load balancing* dengan algoritme *weighted least connection* pada *software defined network*.

6.1 Pengujian Kinerja

Parameter keberhasilan dalam pengujian load balancing dengan algoritme weighted least connection adalah:

1. Sistem dapat melayani *request* yang telah diminta oleh *client*.
2. *Request* dari *client* diteruskan ke *web* server oleh *controller*.
3. *Web* server dengan nilai *weight* yang besar dapat menampung koneksi lebih banyak, begitupun sebaliknya.

Untuk mendapatkan hasil dari pengujian, maka diuji dengan beberapa skenario.

6.1.2 Skenario 1

Pada skenario ini dilakukan pengujian ketika *client* mengakses halaman html dengan macam jumlah koneksi per detik atau disebut dengan *rate*. Dasar dari pemberian jumlah koneksi dan *rate* yaitu berdasarkan pada jumlah koneksi ke berapa server dapat menampung *request* dari *client* tanpa terjadinya *error*. Pengujian ini bertujuan untuk mendapatkan hasil *connection rate, response time* dan *throughput* saat sistem diberi koneksi dengan jumlah yang ditentukan. Pengujian ini menggunakan *tool httperf*.

Langkah pengujian:

1. Menjalankan sistem *load balancing* dengan menggunakan *controller* POX.
2. Menjalankan mininet untuk membuat topologi.
3. Pada pengujian ini terdapat 3 *web* server yang masing-masing memiliki IP: 10.0.0.1, 10.0.0.2, dan 10.0.0.3. Untuk nilai *weight* pada *web* server berturut-turut 1,3,6. Dasar dari pemberian nilai *weight* yaitu spesifikasi *web* server 3 lebih tinggi dibanding *web* server 1 dan 2.
4. Jumlah *request* pada pengujian ini 3000 dan macam jumlah koneksi per detik yaitu 300, 150, dan 75.
5. Pengujian dilakukan sebanyak 5 kali untuk setiap macam jumlah koneksi per detik.
6. *Request* dari client diteruskan oleh controller ke 3 *web* server yang sudah terhubung.

7. Pengambilan data dilakukan setelah *request* dari *client* diterima oleh *web* server. Setelah data diterima selanjutnya akan dilakukan analisis.
8. Data yang diambil dari pengujian ini adalah *response time* dan *throughput.*

Tabel 6.1 Server Menerima Koneksi 3000 dengan *Rate* 75 Koneksi/Detik

No	*Response Time* (ms)	*Throughput* (KB/s)	*Connection Rate* (Conn/s)
1	2,2	228,8	74,9
2	2,4	228,7	74,3
3	2,7	228,9	73,6
4	2	228,8	72,6
5	2,4	228,9	74,6
Rata-Rata	2,34	228,82	74

Pada Tabel 6.1 menunjukkan bahwa data setelah dilakukan pengujian sebanyak 5 kali dengan koneksi 3000 dan *rate* 75 koneksi/detik. Hasil dari parameter *response time* mendapatkan rata-rata sebesar 2,34 ms, *throughput* mendapatkan rata-rata sebesar 228,82 KB/s, dan *connection rate* mendapatkan rata-rata sebesar 74 Conn/s. Dari hasil tersebut menunjukkan bahwa server dapat menangani 3000 koneksi dengan rate 75 koneksi/detik secara baik.

Tabel 6.2 Server Menerima Koneksi 3000 dengan *Rate* 150 Koneksi/Detik

No	*Response Time* (ms)	*Throughput* (KB/s)	*Connection Rate* (Conn/s)
1	2,3	457,4	147,2
2	2,2	456,5	148,6
3	2,1	457,3	149,3
4	2,4	456,7	146,5
5	2,9	456,2	148,6
Rata-Rata	2,38	456,82	148,04

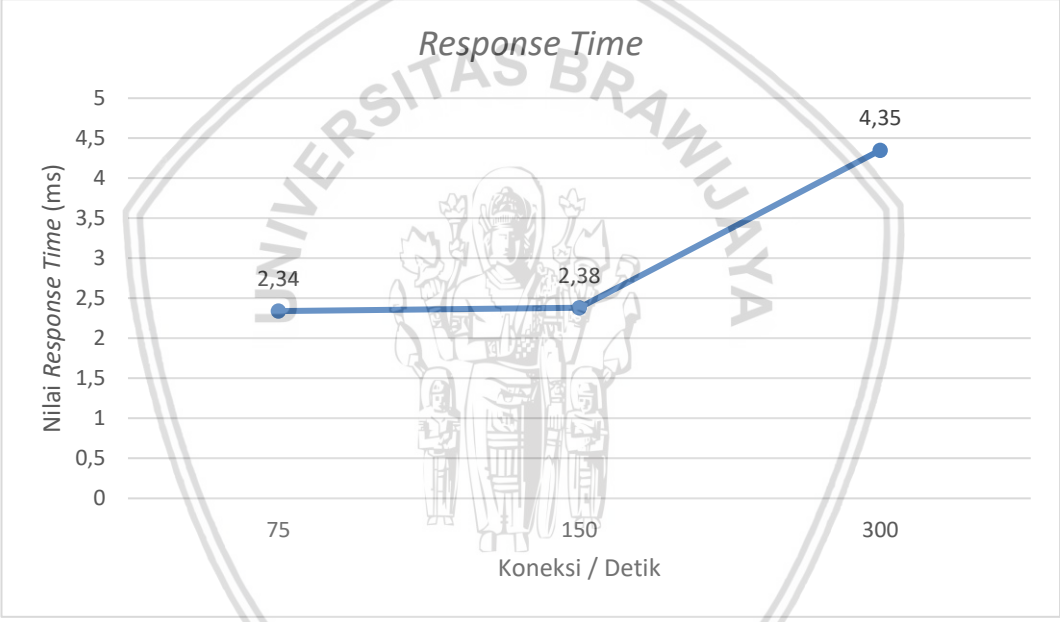
Pada Tabel 6.2 menunjukkan bahwa data setelah dilakukan pengujian sebanyak 5 kali dengan koneksi 3000 dan *rate* 150 koneksi/detik. Hasil dari parameter *response time* mendapatkan rata-rata sebesar 2,38 ms, *throughput* mendapatkan rata-rata sebesar 456,82 KB/s, dan *connection rate* mendapatkan rata-rata sebesar 148,04 Conn/s. Dari hasil tersebut menunjukkan bahwa server dapat menangani 3000 koneksi dengan rate 150 koneksi/detik secara baik.

Tabel 6.3 Server Menerima Koneksi 3000 dengan *Rate* 300 Koneksi/Detik

No	*Response Time* (ms)	*Throughput* (KB/s)	*Connection Rate* (Conn/s)
1	2,6	906,1	296,6

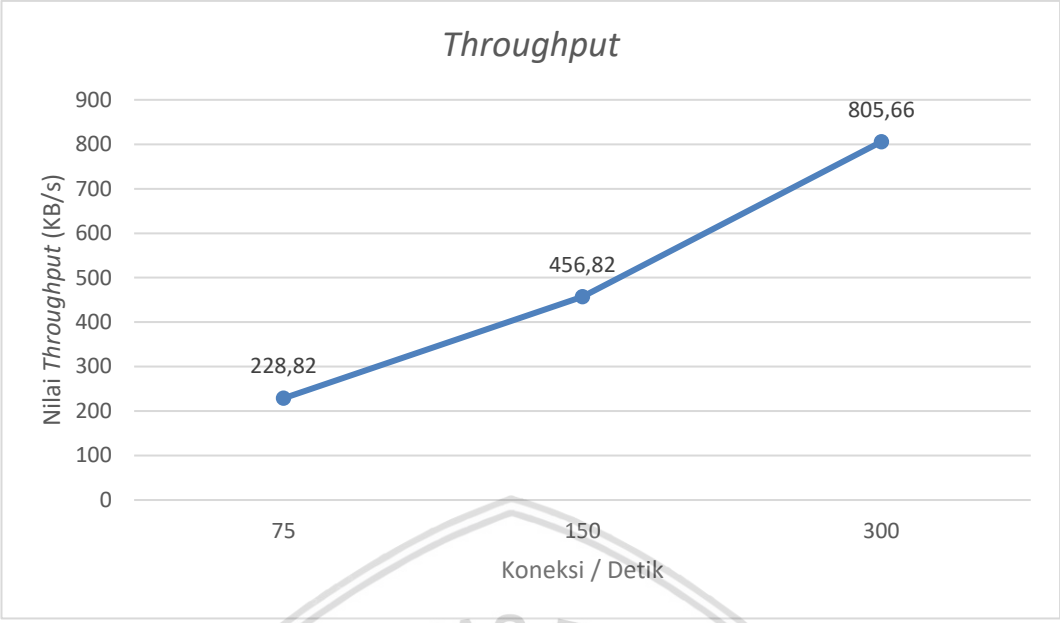
2	4	875,6	295,3
3	4,0	875,4	298,4
4	3,6	760,8	296,8
5	7,2	610,4	294,3
Rata-Rata	4,35	805,66	296,28

Pada Tabel 6.3 menunjukkan bahwa data setelah dilakukan pengujian sebanyak 5 kali dengan koneksi 3000 dan *rate* 300 koneksi/detik. Hasil dari parameter *response time* mendapatkan rata-rata sebesar 4,35 ms, *throughput* mendapatkan rata-rata sebesar 805,66 KB/s, dan *connection rate* mendapatkan rata-rata sebesar 296,28 Conn/s. Dari hasil tersebut menunjukkan bahwa server dapat menangani 3000 koneksi dengan rate 300 koneksi/detik secara baik.



Gambar 6.1 Grafik Perbandingan Nilai *Response Time* dengan Jumlah Koneksi 3000 dan Rate 75, 150, dan 300 Koneksi/Detik

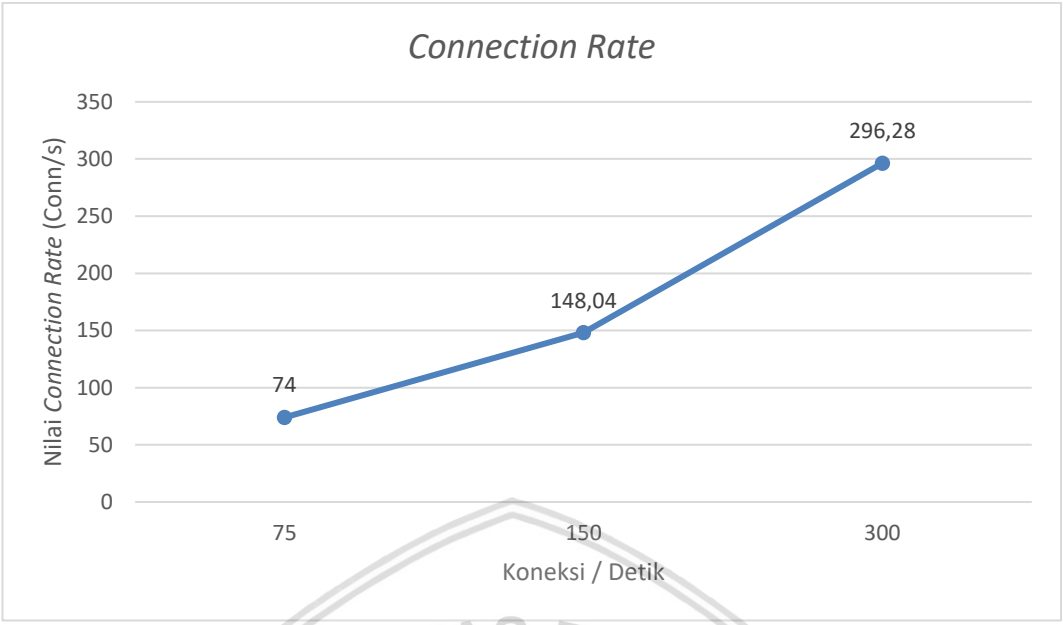
Pada Gambar 6.1 menunjukkan perbandingan nilai *response time* dengan variasi koneksi/detik atau *rate*. Ketika pengujian dilakukan dengan *rate* 75 koneksi per detik memperoleh nilai *response time* 2,34 ms, pada pengujian 150 koneksi/detik memperoleh nilai *response time* 2,38 ms dan pengujian 300 koneksi/detik memperoleh nilai *response time* 4,35 ms. Hasil ini menunjukkan bahwa semakin rendah *rate* (koneksi/detik) maka nilai *response time* yang diberikan akan baik. Terlihat pada *rate* 75 rata-rata nilai *response time* adalah 2,34 ms. Begitu juga sebaliknya *rate* tinggi akan memperlambat *response time*, terlihat pada *rate* 300 rata-rata nilai *response time* adalah 4,35 ms.



Gambar 6.2 Grafik Perbandingan Nilai *Throughput* dengan Jumlah Koneksi 3000 dan Rate 75, 150, dan 300 Koneksi/Detik

Pada Gambar 6.2 menunjukkan perbandingan nilai *throughput* dengan variasi koneksi/detik atau *rate*. Ketika pengujian dilakukan dengan *rate* 75 koneksi per detik memperoleh nilai *throughput* 228,82 KB/s, pada pengujian 150 koneksi/detik memperoleh nilai *throughput* 456,82 KB/s dan pengujian 300 koneksi/detik memperoleh nilai *throughput* 805,66 KB/s. Hasil ini menunjukkan bahwa semakin tinggi *rate* (koneksi/detik) maka nilai *throughput* yang diberikan juga semakin tinggi. Terlihat pada *rate* 300 rata-rata nilai *throughput* adalah 805,66 KB/s. Begitu juga sebaliknya *rate* rendah akan mendapatkan nilai *throughput* yang rendah juga, terlihat pada *rate* 75 rata-rata nilai *throughput* adalah 228,82 KB/s. Tingginya nilai *throughput* menunjukkan bahwa kinerja *load balancer* semakin meningkat.

Pada Gambar 6.3 menunjukkan perbandingan nilai *connection rate* dengan variasi koneksi/detik atau *rate*. Ketika pengujian dilakukan dengan *rate* 75 koneksi per detik memperoleh nilai *connection rate* 74 Conn/s, pada pengujian 150 koneksi/detik memperoleh nilai *connection rate* 148,04 Conn/s dan pengujian 300 koneksi/detik memperoleh nilai *connection rate* 296,28 Conn/s. Hasil ini menunjukkan bahwa semakin tinggi *rate* (koneksi/detik) maka nilai *connection rate* yang diberikan juga semakin tinggi. Terlihat pada *rate* 300 rata-rata nilai *connection rate* adalah 296,28 Conn/s. Begitu juga sebaliknya *rate* rendah akan mendapatkan nilai *connection rate* yang rendah juga, terlihat pada *rate* 75 rata-rata nilai *connection rate* adalah 74 Conn/s.



Gambar 6.3 Grafik Perbandingan Nilai *Connection Rate* dengan Jumlah Koneksi 3000 dan Rate 75, 150, dan 300 Koneksi/Detik

6.1.3 Skenario 2

Pada skenario ini dilakukan pengujian ketika *client* mengakses gambar dengan ukuran data sebesar 2,2 MB dan macam jumlah koneksi per detik atau disebut dengan *rate*. Dasar dari pemberian jumlah koneksi dan *rate* yaitu berdasarkan pada jumlah koneksi ke berapa server dapat menampung *request* dari *client* tanpa terjadinya *error*. Pengujian ini bertujuan untuk mendapatkan hasil *connection rate, response time* dan *throughput* saat sistem diberi koneksi dengan jumlah yang ditentukan. Pengujian ini menggunakan *tool httperf*.

Langkah pengujian:

1. Menjalankan sistem *load balancing* dengan menggunakan *controller* POX.
2. Menjalankan mininet untuk membuat topologi.
3. Pada pengujian ini terdapat 3 *web* server yang masing-masing memiliki IP: 10.0.0.1, 10.0.0.2, dan 10.0.0.3. Untuk nilai *weight* pada *web* server berturut-turut 1,3,6. Dasar dari pemberian nilai *weight* yaitu spesifikasi *web* server 3 lebih tinggi dibanding *web* server 1 dan 2.
4. Jumlah *request* pada pengujian ini 500 dan macam jumlah koneksi per detik yaitu 50, 25, dan 13.
5. Pengujian dilakukan sebanyak 5 kali untuk setiap macam jumlah koneksi per detik.
6. *Request* dari *client* diteruskan oleh *controller* ke 3 *web* server yang sudah terhubung.

7. Pengambilan data dilakukan setelah *request* dari *client* diterima oleh *web* server. Setelah data diterima selanjutnya akan dilakukan analisis.
8. Data yang diambil dari pengujian ini adalah *response time* dan *throughput.*

Tabel 6.4 Server Menerima Koneksi 500 dengan Rate 13 Koneksi/Detik

No	*Response Time* (ms)	*Throughput* (KB/s)	*Connection Rate* (Conn/s)
1	1,8	27517,3	13
2	1,2	27545,6	13
3	0,9	27548,2	12,8
4	2,3	27521,7	12,6
5	3,3	27514,5	13
Rata-Rata	1,9	27529,46	12,88

Pada Tabel 6.4 menunjukkan bahwa data setelah dilakukan pengujian sebanyak 5 kali dengan koneksi 500 dan *rate* 13 koneksi/detik. Hasil dari parameter *response time* mendapatkan rata-rata sebesar 1,9 ms, *throughput* mendapatkan rata-rata sebesar 27529,46 KB/s, dan *connection rate* mendapatkan rata-rata sebesar 12,88 Conn/s. Dari hasil tersebut menunjukkan bahwa server dapat menangani 500 koneksi dengan rate 13 koneksi/detik secara baik.

Tabel 6.5 Server Menerima Koneksi 500 dengan Rate 25 Koneksi/Detik

No	*Response Time* (ms)	*Throughput* (KB/s)	*Connection Rate* (Conn/s)
1	2	52897,6	25
2	2,7	52968	24,8
3	5,4	52930,6	24,5
4	1,5	52899,3	25
5	5,2	52966,8	24,7
Rata-Rata	3,36	52932,46	24,8

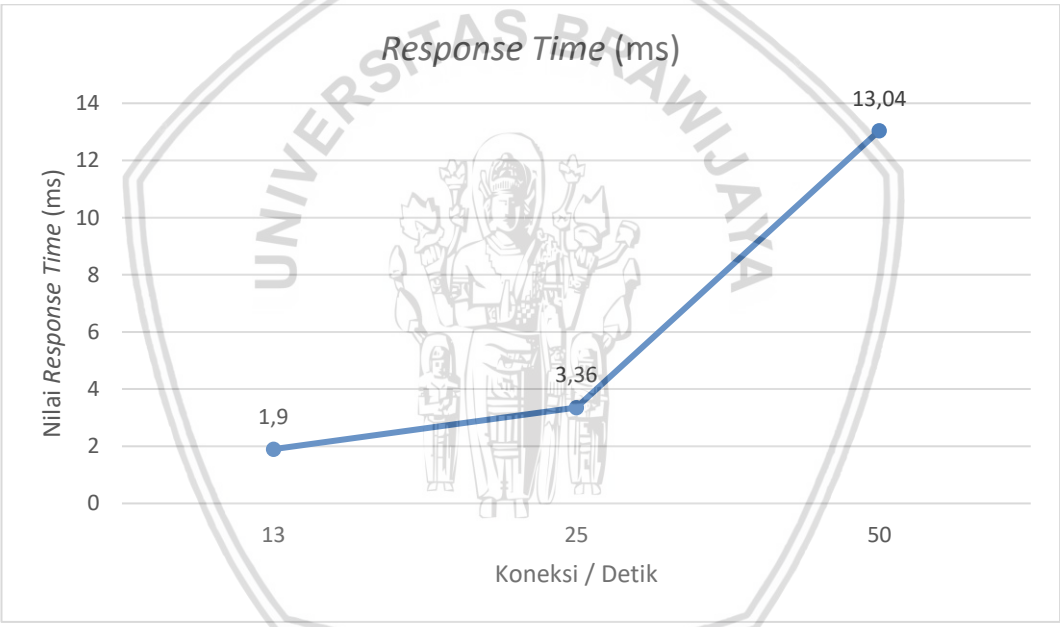
Pada Tabel 6.5 menunjukkan bahwa data setelah dilakukan pengujian sebanyak 5 kali dengan koneksi 500 dan *rate* 25 koneksi/detik. Hasil dari parameter *response time* mendapatkan rata-rata sebesar 3,36 ms, *throughput* mendapatkan rata-rata sebesar 52932,46 KB/s, dan *connection rate* mendapatkan rata-rata sebesar 24,8 Conn/s. Dari hasil tersebut menunjukkan bahwa server dapat menangani 500 koneksi dengan rate 25 koneksi/detik secara baik.

Tabel 6.6 Server Menerima Koneksi 500 dengan Rate 50 Koneksi/Detik

No	*Response Time* (ms)	*Throughput* (KB/s)	*Connection Rate* (Conn/s)
1	17,9	105297,8	49,7

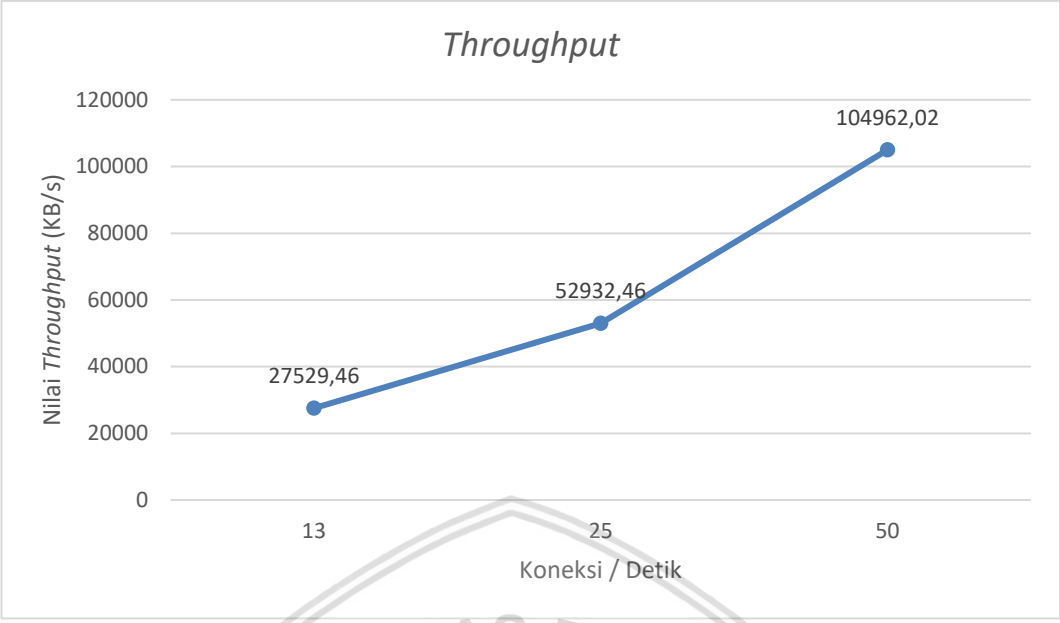
2	13,6	105397,2	48,5
3	10,5	105146,3	49,6
4	13	105401,8	49,3
5	10,2	103567	48,6
Rata-Rata	13,04	104962,02	49,14

Pada Tabel 6.6 menunjukkan bahwa data setelah dilakukan pengujian sebanyak 5 kali dengan koneksi 500 dan *rate* 50 koneksi/detik. Hasil dari parameter *response time* mendapatkan rata-rata sebesar 13,04 ms, *throughput* mendapatkan rata-rata sebesar 104962,02 KB/s, dan *connection rate* mendapatkan rata-rata sebesar 49,14 Conn/s. Dari hasil tersebut menunjukkan bahwa server dapat menangani 500 koneksi dengan rate 50 koneksi/detik secara baik.



Gambar 6.4 Grafik Perbandingan Nilai *Response Time* dengan Jumlah Koneksi 500 dan Rate 13, 25, dan 50 Koneksi/Detik

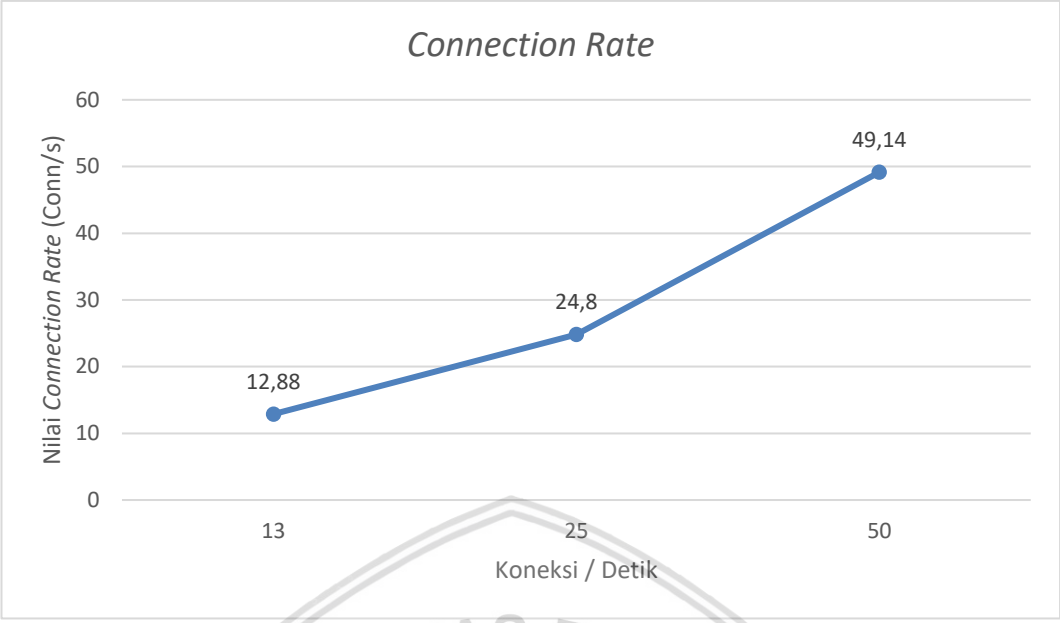
Pada Gambar 6.4 menunjukkan perbandingan nilai *response time* dengan variasi koneksi/detik atau *rate*. Ketika pengujian dilakukan dengan *rate* 13 koneksi per detik memperoleh nilai *response time* 1,9 ms, pada pengujian 25 koneksi/detik memperoleh nilai *response time* 3,36 ms dan pengujian 300 koneksi/detik memperoleh nilai *response time* 13,04 ms. Hasil ini menunjukkan bahwa semakin rendah *rate* (koneksi/detik) maka nilai *response time* yang diberikan akan baik. Terlihat pada *rate* 13 rata-rata nilai *response time* adalah 1,9 ms. Begitu juga sebaliknya *rate* tinggi akan memperlambat *response time*, terlihat pada *rate* 50 rata-rata nilai *response time* adalah 13,04 ms.



Gambar 6.5 Grafik Perbandingan Nilai *Throughput* dengan Jumlah Koneksi 500 dan Rate 13, 25, dan 50 Koneksi/Detik

Pada Gambar 6.5 menunjukkan perbandingan nilai *throughput* dengan variasi koneksi/detik atau *rate*. Ketika pengujian dilakukan dengan *rate* 13 koneksi per detik memperoleh nilai *throughput* 27529,46 KB/s, pada pengujian 25 koneksi/detik memperoleh nilai *throughput* 52932,46 KB/s dan pengujian 50 koneksi/detik memperoleh nilai *throughput* 104962,02 KB/s. Hasil ini menunjukkan bahwa semakin tinggi *rate* (koneksi/detik) maka nilai *throughput* yang diberikan juga semakin tinggi. Terlihat pada *rate* 50 rata-rata nilai *throughput* adalah 104962,02 KB/s. Begitu juga sebaliknya *rate* rendah akan mendapatkan nilai *throughput* yang rendah juga, terlihat pada *rate* 13 rata-rata nilai *throughput* adalah 27529,46 KB/s. Tingginya nilai *throughput* menunjukkan bahwa kinerja *load balancer* semakin meningkat.

Pada Gambar 6.6 menunjukkan perbandingan nilai *connection rate* dengan variasi koneksi/detik atau *rate*. Ketika pengujian dilakukan dengan *rate* 13 koneksi per detik memperoleh nilai *connection rate* 12,88 Conn/s, pada pengujian 25 koneksi/detik memperoleh nilai *connection rate* 24,8 Conn/s dan pengujian 50 koneksi/detik memperoleh nilai *connection rate* 49,14 Conn/s. Hasil ini menunjukkan bahwa semakin tinggi *rate* (koneksi/detik) maka nilai *connection rate* yang diberikan juga semakin tinggi. Terlihat pada *rate* 50 rata-rata nilai *connection rate* adalah 49,14 Conn/s. Begitu juga sebaliknya *rate* rendah akan mendapatkan nilai *connection rate* yang rendah juga, terlihat pada *rate* 13 rata-rata nilai *connection rate* adalah 12,88 Conn/s.



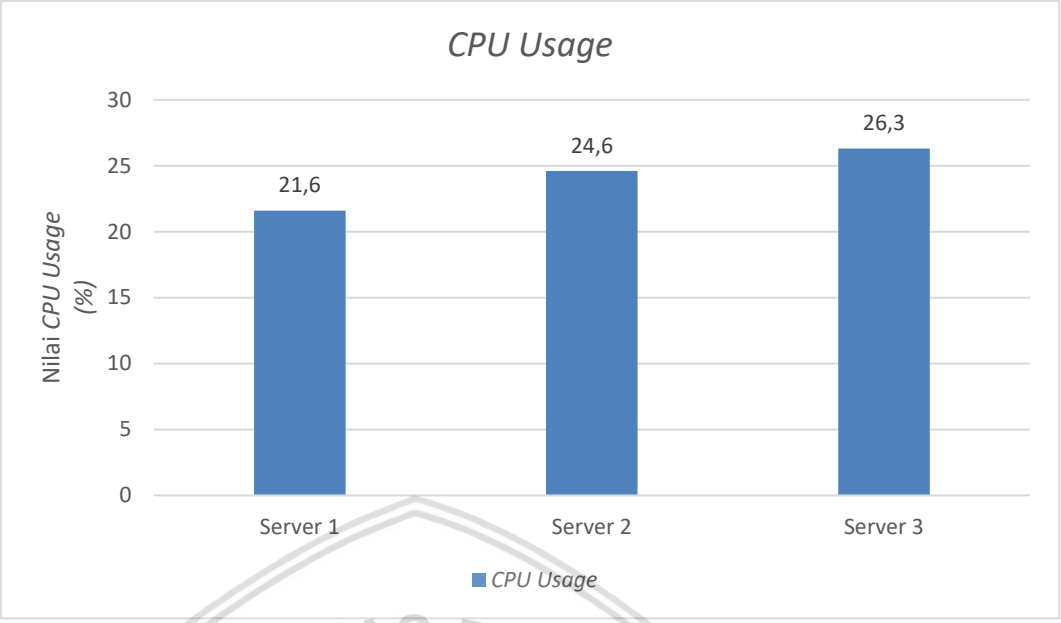
Gambar 6.6 Grafik Perbandingan Nilai *Connection Rate* dengan Jumlah Koneksi 500 dan Rate 13, 25, dan 50 Koneksi/Detik

6.1.4 Skenario 3

Pada skenario pengujian ini akan melakukan pengujian CPU *usage* untuk melihat penggunaan CPU dari setiap server. Pengujian ini dilakukan percobaan sebanyak 5 kali dengan request dari *client* sebanyak 3000 dan *rate* 300.

Langkah pengujian:

1. Menjalankan sistem *load balancing* dengan menggunakan *controller* POX.
2. Menjalankan mininet untuk membuat topologi.
3. Pada pengujian ini terdapat 3 *web* server yang masing-masing memiliki IP: 10.0.0.1, 10.0.0.2, dan 10.0.0.3. Untuk nilai *weight* pada *web* server berturut-turut 1,3,6. Dasar dari pemberian nilai *weight* yaitu spesifikasi *web* server 3 lebih tinggi dibanding web server 1 dan 2.
4. Jumlah *request* pada pengujian ini 1000 dan macam jumlah koneksi per detik yaitu 100.
5. Pengujian ini menggunakan *tool httperf* untuk melihat cpu *usage*.
6. Data yang diambil dari pengujian ini yaitu cpu *usage*



Gambar 6.7 Grafik Perbandingan CPU *Usage*

Dari Gambar 6.5 menunjukkan bahwa penggunaan CPU pada server 1 nilai rata-rata yang didapat yaitu 21,6%, server 2 mendapatkan nilai rata-rata 24,6% dan server 3 mendapatkan nilai rata-rata 26,3%. Dari hasil pengujian diatas menunjukan server 3 penggunaan CPU-nya lebih tinggi dibanding server 1 dan 2. Hal ini dikarenakan server 3 dirancang untuk dapat menampung koneksi lebih banyak. Semakin banyak koneksi yang ditampung maka semakin banyak juga penggunaan CPU yang digunakan.

BAB 7 PENUTUP

7.1 Kesimpulan

Berdasarkan hasil dari perancangan, implementasi, pengujian, dan analisis yang telah dilakukan pada *load balancing* di *web* server menggunakan algoritme *weighted least connection*, maka dapat disimpulkan yaitu:

1. Pada implementasi algoritme *load balancing*, *controller* dijadikan sebagai *load balancer* yang menerapkan algoritme *weighted least connection*. Untuk mengetahui jumlah koneksi yang dimiliki oleh server, penelitian ini menggunakan informasi *flow* yang tersimpan pada perangkat *switch*. Dari implementasi tersebut, *load balancing* menggunakan algoritme *weighted least connection* pada *software defined network* dapat berjalan dengan baik serta mampu mendistribusikan trafik ke beberapa server di dalam cluster. Algoritme *weighted least connection* dapat membagi beban *traffic* berdasarkan bobot kinerja dari setiap server. Penentuan bobot kinerja ditentukan oleh administrator dengan melihat spesifikasi dari server yang digunakan. Server yang memiliki spesifikasi tinggi akan diberikan nilai bobot yang tinggi, sedangkan server yang memiliki spesifikasi rendah akan diberikan nilai bobot yang rendah. Nilai bobot/*weight* sangat berpengaruh dalam menampung jumlah koneksi oleh suatu server. Server yang memiliki spesfikasi yang lebih unggul dari server lainnya maka akan diberi nilai *weight* lebih besar agar dapat menampung koneksi yang lebih banyak.
2. Pada pengujian sistem yang dilakukan pada penelitan ini terdiri dari beberapa skenario diantaranya mengukur nilai *connection rate*, *response time, throughput,* dan CPU *usage*. Pada skenario ini mengukur nilai *connection rate*, *response time,* dan *throughput* dengan memberikan *request* dari *client* sebanyak 3000 dengan *rate* 75, 150, 300. Pada pengujian yang dilakukan yaitu mengakses halaman index html. Hasilnya untuk *connection rate* tertinggi yaitu pada pengujian dengan *rate* 300 sebesar 296,28 Conn/s, sedangkan pada pengujian dengan *rate* 75 memperoleh nilai *connection rate* terkecil yaitu 74 Conn/s. Nilai *connection rate* ditentukan dari *rate,* semakin tinggi *rate* maka nilai *connection rate* juga semakin tinggi. Untuk nilai *response time* terkecil yaitu pada pengujian dengan *rate* 75 yaitu 2,34 ms, sedangkan pada pengujian dengan *rate* 300 memperoleh nilai *response time* terbesar sebesar 4,35 ms. Hasil ini menunjukkan bahwa semakin rendah *rate* (koneksi/detik) maka nilai *response time* yang diberikan akan baik. Untuk nilai *throughput* tertinggi yaitu pada pengujian dengan *rate* 300 sebesar 805,66 KB/s, sedangkan pada pengujian dengan *rate* 75 memperoleh nilai *throughput* terkecil yaitu 228,82 KB/s. Nilai *throughput* ditentukan dari *rate* semakin besar rate maka semakin besar juga nilai *throughput*.

Lalu skenario berikutnya mengukur penggunaan CPU dari setiap server. Hasilnya dari pengujian yang dilakukan menunjukan pada server 3 memiliki nilai CPU *usage* tertinggi yaitu 26,3% sedangkan server 1 memiliki nilai CPU *usage* terendah yaitu 21,6%. Hal ini dikarenakan server 3 dirancang untuk dapat menampung koneksi lebih banyak. Semakin banyak koneksi yang ditampung maka semakin banyak juga penggunaan CPU yang digunakan.

7.2 Saran

Saran yang dianjurkan untuk pengembangan penelitian selanjutnya terhadap penerapan *load balancing* dengan algoritme *weighted least connection* pada *web* server yaitu:

1. Melakukan pengujian dan analisa dengan parameter yang lebih banyak lagi, tidak hanya *connection rate*, *response time, throughput,* dan CPU *usage* saja.
2. Melakukan pengujian pada *load balancing* dengan jumlah *request* yang lebih banyak lagi.
3. Membandingkan hasil pengujian dan analisa antara algoritme *weighted least connection* dengan algoritme lainnya.

DAFTAR PUSTAKA

Angsar, N., 2014. Pengujian Distribusi Beban Web dengan Algoritma Least Connection dan Weighted Least Connection. Volume 3.

Astuto, B. et al., 2014. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *IEEE Communications Surveys & Tutorials,* 16(3).

Golinelli, E. S., 2015. *A Software Defined Networking Evaluation Approach to Distrubting Load Using POX, Floodlight and BIG-IP 1600.* Oslo: University of Oslo.

Kaur, H. & Jyoti, N., 2017. Traffic Based Load Balancing in Software. *International Journal on Computer Science and Engineering (IJCSE),* Volume 9.

Kaur, S., Singh, J. & Ghumman, N. S., 2014. Network Programmability Using POX Controller. *International Conference Communication, Computing & Systems,* Volume 1.

Keti, F. & Askar, S., 2015. Emulation of Software Defined Networks Using Mininet in Different Simulation Environments. *2015 6th International Conference onn Intelligent Systems Modelling and Simulation.*

Kreutz, D. et al., 2014. Software Defined Networking: A Comprehensive Survey. *Procedings of the IEEE,* Volume 103.

Lee, K.-H., 2014. Mobility Management Framework in Software Defined Networks. *International Journal of Software Engineering and Its Applications,* Volume 8, pp. 1-10.

Paul Göransson, C. B., 2014. *Software Defined Networks: A Comprehensive Approach.* Waltham, USA: s.n.

Rahmana, D., 2017. *Analisis Load Balancing Pada Web Server Menggunakan Algoritme Least Connection.* Malang: Universitas Brawijaya.

Rahman, M., Iqbal, S. & Gao, J., 2014. Load Balancer as a Service in Cloud Computing. *2014 IEEE 8th International Symposium on Service Oriented System Engineering.*

Red Hat, I., 2014. *Red Hat Enterprise Linux 5.* [Online] Available at: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/5/html/Virtual_Server_Administration/index.html [Diakses 12 September 2017].

Sabiya & Singh, J., 2016. Weighted Round-Robin Load Balancing Using Software Defined Network. *International Journal of Advanced Research in Computer Science and Software Engineering,* 6(6), pp. 621-625.

Singh, H. & Kumar, D. S., 2011. Dispatcher Based Dynamic Load Balancing on Web Server System. *International Journal of Grid and Distributed Computing,* Volume 4, pp. 89-106.

Supramana & Prismana, I. G. L. P. E., 2016. Implementasi Load Balancing Pada Web Server Dengan Menggunakan Apache. Volume 5, pp. 117-125.

Tasneem, S. & Ammar, R., 2012. Performance Study of a Distributed Web Server: An Analytical Approach. *Journal of Software Engineering and Applications*.

Zhong, H. et al., 2015. An Efficient SDN Load Balancing Scheme Based on. *Mobile Information Systems.*